

A PRACTICAL REPORT
ON
COMPUTER VISION

SUBMITTED BY
Mr. Prasad Yash Abhay Urmila

UNDER THE GUIDANCE OF
Mrs. Hina Mhmd

Submitted in fulfillment of the requirements for qualifying
M.Sc.IT Part-1 Semester-2 Examination 2025-2026

University of Mumbai
Department of Information Technology

University of Mumbai

University of Mumbai



Institute of Distance and Open Learning (IDOL)

Dr. Shankardayal Sharma bhavan, Vidyanagari, Santacruz(E)

PCP CENTER: RIZVI COLLEGE, BANDRA (W)

Certificate

*This is to certify that **Computer vision Practical** performed at RIZVI COLLEGE, BANDRA (W) by Mr. **PRASAD YASH ABHAY** holding Seat No/Application No.8166083 studying Masters of Science in Information Technology Semester – 2 has been satisfactorily completed as prescribed by the University of Mumbai, during the year 2025 – 2026.*

Subject In-Charge

Coordinator In-Charge

External Examiner

Index

Sr. No	Title	Date	Signature
1	Perform Geometric transformation.		
	A. Image Scaling.		
	B. Image Shrinking		
	C. Image Rotation		
	D. Affine Transformation		
	E. Perspective Transformation		
	F. Shearing X-axis		
	G. Shearing Y-axis		
	H. Reflected Image		
	I. Cropped Image		
2	Perform Image Stitching.		
3	Perform Camera Calibration.		
4	A. Perform the following Face detection.		
	B. Perform the following: i. Object detection ii. Line Detection iii. Hough transform		
	C. Perform the following Pedestrian detection.		
	D. Perform the following Face Recognition		
5	A. Implement object detection and tracking from video.		
	B. Implement object detection and tracking from video. (Count number of Faces using Python)		
	C. Implement object detection and tracking from video. (Object Tracking using Homography)		
6	Perform Colorization.		
7	Perform Text Detection and Recognition.		
8	Construct 3D model from Images.		
9	Perform Feature extraction using RANSAC.		

Practical 1

Aim: Perform Geometric transformation.

Theory:

Geometric Transformations:

Geometric transformation is a fundamental technique used in image processing that involves manipulating the spatial arrangement of pixels in an image. It is used to modify the geometric properties of an image, such as its size, shape, position, and orientation.

OpenCV provides two transformation functions, `cv2.warpAffine` and `cv2.warpPerspective`, with which you can have all kinds of transformations. `Cv2.warpAffine` takes a 2x3 transformation matrix while `cv2.warpPerspective` takes a 3x3 transformation matrix as input.

Translation

Translation is the shifting of object's location. If you know the shift in (x,y) direction, let it be , you can create the transformation matrix as follows:

$$M = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}$$

You can take make it into a Numpy array of type `np.float32` and pass it into `cv2.warpAffine()` function. See below example for a shift of (100,50):

Warning:

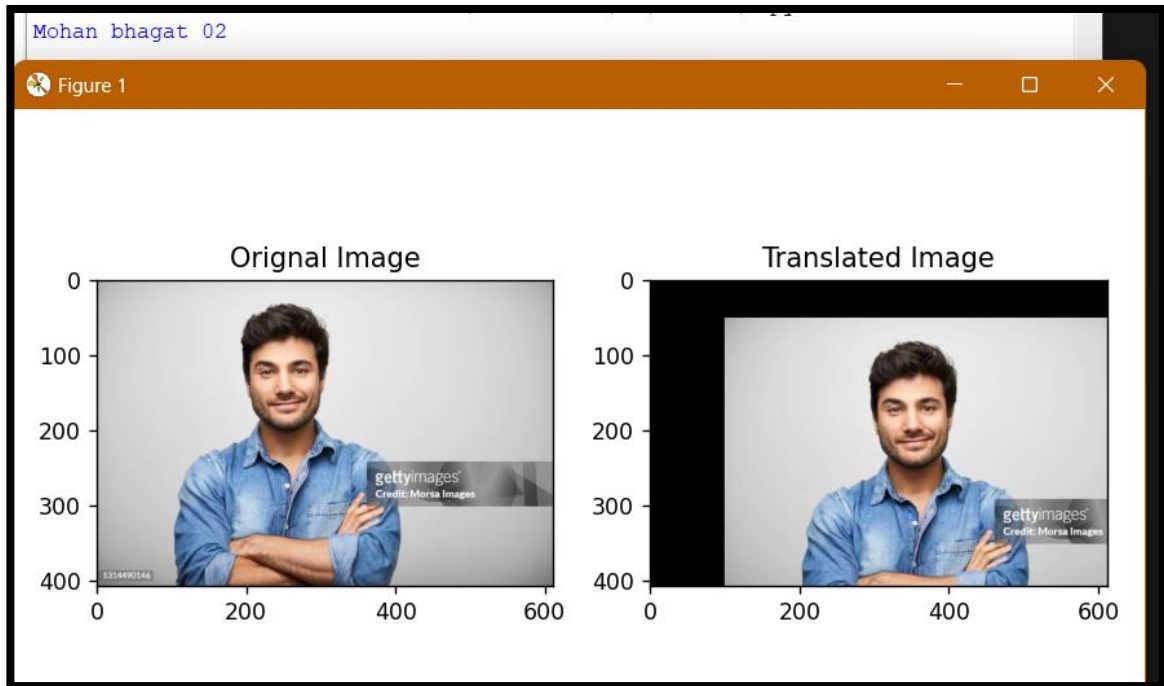
Third argument of the `cv2.warpAffine()` function is the size of the output image, which should be in the form of (width, height). Remember width = number of columns, and height = number of rows.

Code:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
img = cv2.imread('1.jpg')
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, channels = img_rgb.shape
M = np.float32([[1,0,100],[0,1,50]])
dst = cv2.warpAffine(img_rgb,M ,(cols,rows))
fig, axs = plt.subplots(1, 2, figsize=(7, 4))
axs[0].imshow(img_rgb)
axs[0].set_title('Original Image')
axs[1].imshow(dst)
axs[1].set_title('Translated Image')
print("Mohan bhagat 02")
```

```
plt.tight_layout()
plt.show()
```

Output:



Practical -1(A)

Aim: Image Scaling.

Theory:

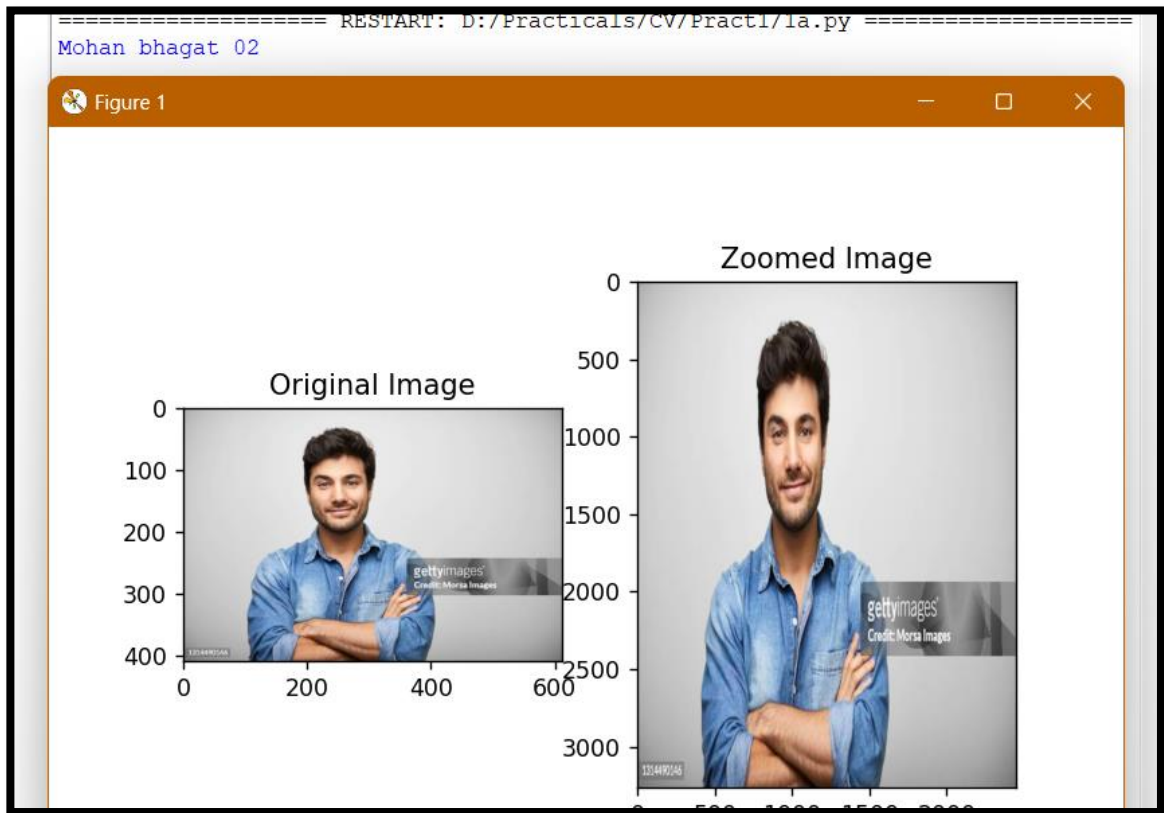
Scaling

Scaling is just resizing of the image. OpenCV comes with a function `cv2.resize()` for this purpose. The size of the image can be specified manually, or you can specify the scaling factor. Different interpolation methods are used. Preferable interpolation methods are `cv2.INTER_AREA` for shrinking and `cv2.INTER_CUBIC` (slow) & `cv2.INTER_LINEAR` for zooming. By default, interpolation method used is `cv2.INTER_LINEAR` for all resizing purposes. You can resize an input image either of following methods:

Code:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
print("Mohan bhagat 02")
img=cv2.imread('1.jpg')
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, channels= img_rgb.shape
resize_img=cv2.resize( img_rgb, (0,0), fx=4, fy=8, interpolation=cv2.INTER_CUBIC)
plt.subplot (121), plt.imshow(img_rgb), plt.title("Original Image")
plt.subplot (122), plt.imshow(resize_img), plt.title('Zoomed Image')
plt.show()
```

Output:



Practical – 1(B)

Aim: Image Shrinking.

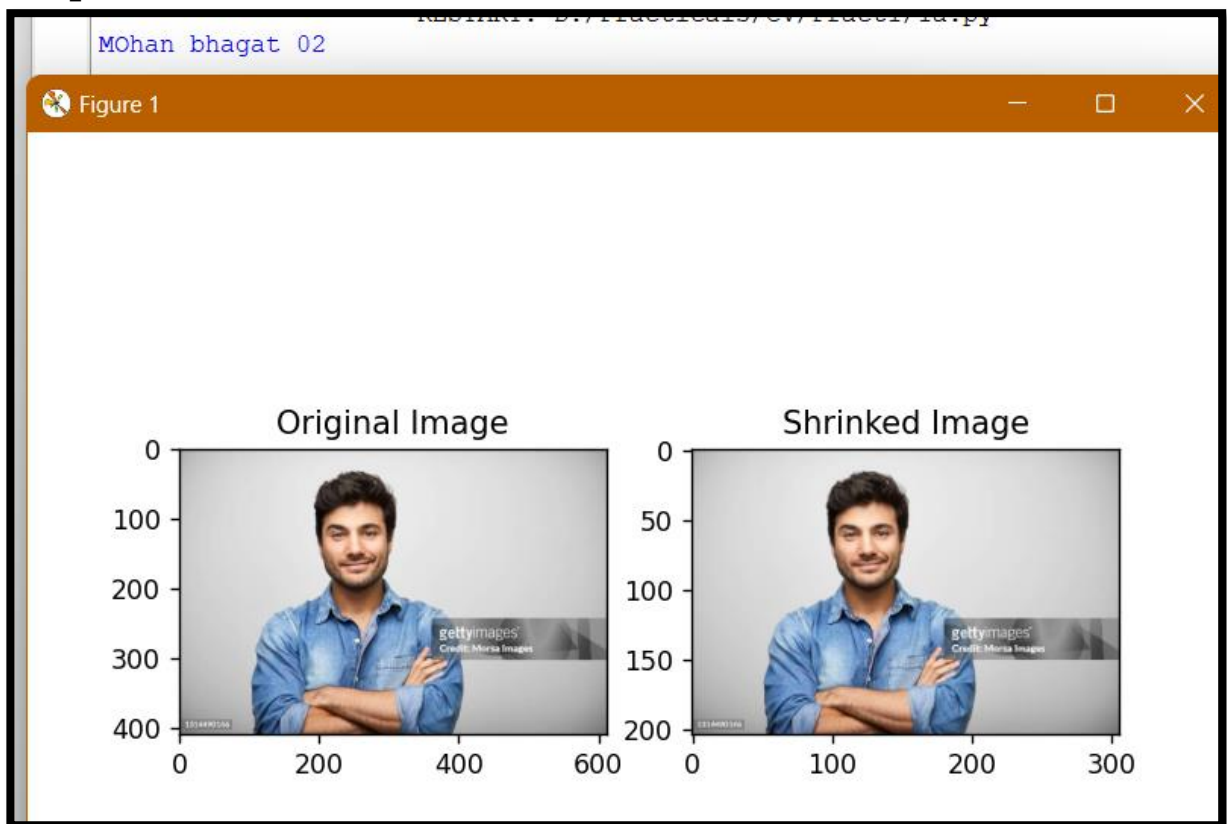
Theory:

Shrinking

Shrinking Image shrinking in Python involves reducing the dimensions of an image, effectively making it smaller while maintaining its aspect ratio. This process is commonly done using libraries like PIL (Python Imaging Library) or its fork, Pillow. By resizing the image to smaller dimensions, either by specifying new dimensions or a scaling factor, you can achieve the desired reduction in size. This is useful for tasks like image compression, thumbnail generation, or preparing images for web display, where smaller file sizes are advantageous. The process typically involves opening the image, calculating the new dimensions, resizing the image accordingly, and then saving the resized image to a new file.

Code:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
print("MOhan bhagat 02")
img=cv2.imread('1.jpg')
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, channels= img_rgb.shape
resize_img=cv2.resize (img_rgb, (0,0), fx=0.5, fy=0.5,
interpolation=cv2.INTER_AREA)
plt.subplot (121), plt.imshow(img_rgb), plt.title("Original Image")
plt.subplot (122), plt.imshow(resize_img), plt.title('Shrunked Image')
plt.show()
```

Output:**Practical – 1 (C)**

Aim: Image Rotation.

Theory:

Rotation:

Rotation of an image for an angle θ is achieved by the transformation matrix of the form

$$M = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

But OpenCV provides scaled rotation with adjustable center of rotation so that you can rotate at any location you prefer. Modified transformation matrix is given by

$$\begin{bmatrix} \alpha & \beta & (1 - \alpha) \cdot \text{center.x} - \beta \cdot \text{center.y} \\ -\beta & \alpha & \beta \cdot \text{center.x} + (1 - \alpha) \cdot \text{center.y} \end{bmatrix}$$

where:

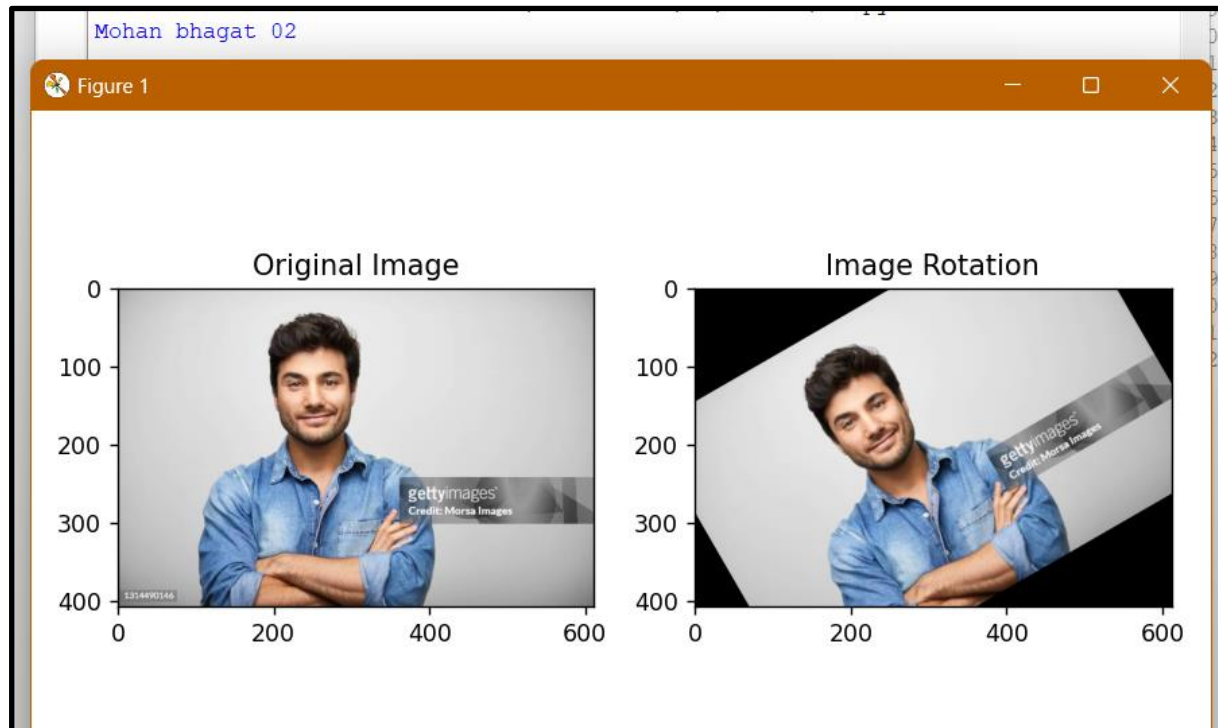
$$\begin{aligned} \alpha &= \text{scale} \cdot \cos \theta \\ \beta &= \text{scale} \cdot \sin \theta \end{aligned}$$

To find this transformation matrix, OpenCV provides a function, **cv2.getRotationMatrix2D**. Check below example which rotates the image by 90 degree with respect to center without any scaling.

Code:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
print("Mohan bhagat 02")
img=cv2.imread('1.jpg')
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, channels= img_rgb.shape
center = (cols // 2, rows // 2)
angle =30
scale = 1
rotation_matrix = cv2.getRotationMatrix2D (center, angle, scale)
rotated_image = cv2.warpAffine (img_rgb, rotation_matrix, (cols, rows))
fig, axs = plt.subplots (1, 2, figsize=(7, 4))
axs[0].imshow(img_rgb)
axs[0].set_title("Original Image")
axs [1].imshow(rotated_image)
axs[1].set_title('Image Rotation')
plt.tight_layout()
plt.show()
```

Output:



Practical – 1 (D)

Aim: Affine Transformation.

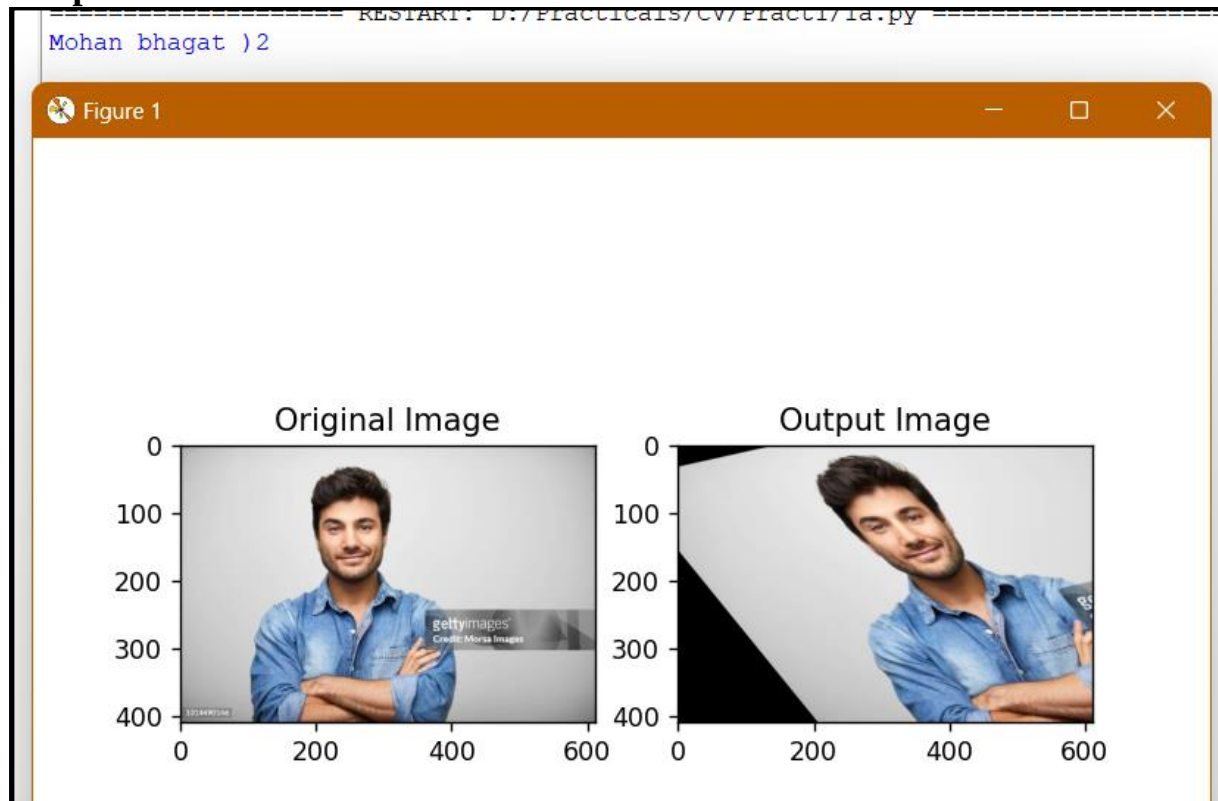
Theory:

In affine transformation, all parallel lines in the original image will still be parallel in the output image. To find the transformation matrix, we need three points from input image and their corresponding locations in output image. Then **cv2.getAffineTransform** will create a 2x3 matrix which is to be passed to **cv2.warpAffine**.

Code:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
print('Mohan bhagat )2')
img = cv2.imread('1.jpg')
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, channels = img_rgb.shape
pts1 = np.float32([[50,50],[200,50],[50,200]])
pts2 = np.float32([[20,90],[200,50],[150,250]])
M = cv2.getAffineTransform(pts1, pts2)
dst = cv2.warpAffine(img_rgb,M,(cols,rows))
plt.subplot(121),plt.imshow(img_rgb), plt.title("Original Image")
plt.subplot(122),plt.imshow(dst), plt.title("Output Image")
plt.show()
```

Output:



Practical – 1 (E)

Aim: Perspective Transformation.

Theory:

Perspective Transformation:

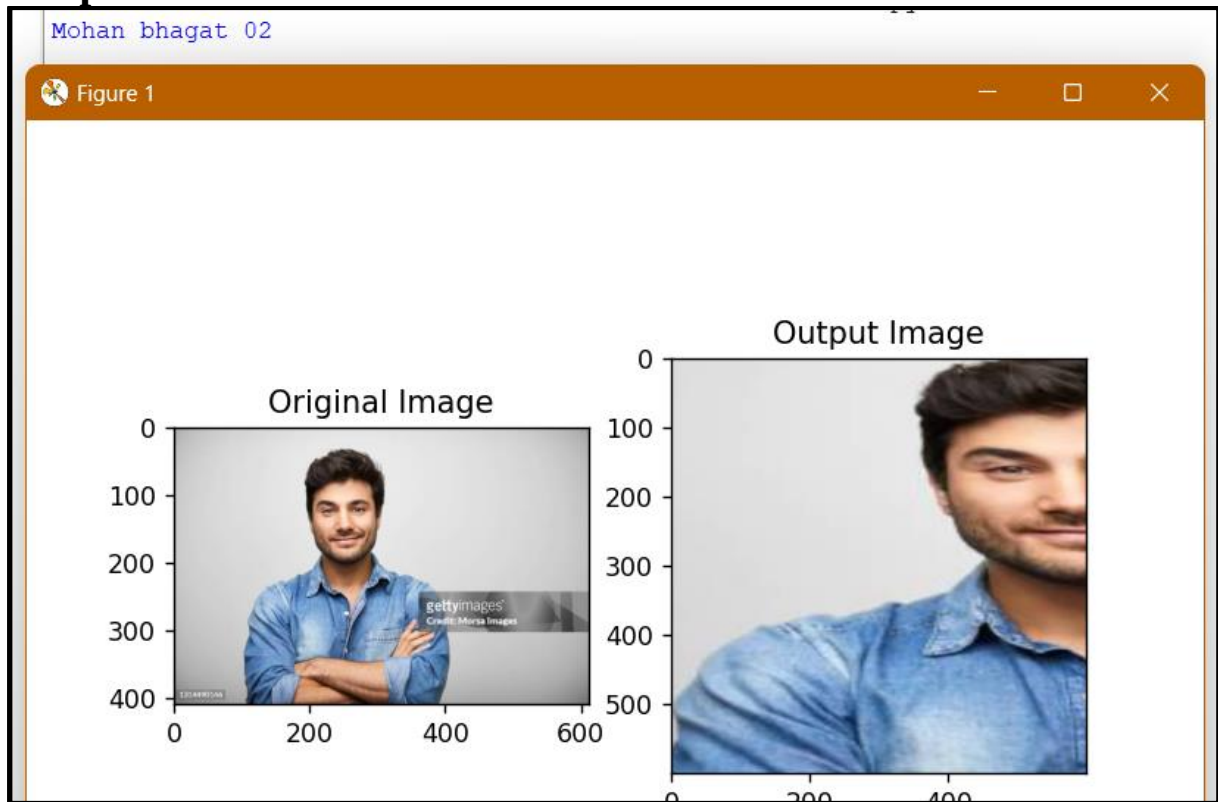
For perspective transformation, you need a 3x3 transformation matrix. Straight lines will remain straight even after the transformation. To find this transformation matrix, you need 4 points on the input image and corresponding points on the output image. Among these 4 points, 3 of them should not be collinear. Then transformation matrix can be found by the function **cv2.getPerspectiveTransform**. Then apply **cv2.warpPerspective** with this 3x3 transformation matrix.

Code:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
print('Mohan bhagat 02')
img = cv2.imread('1.jpg')
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, channels = img_rgb.shape
```

```
pts1 = np.float32([[100,51],[260,37],[110,325], [250,320]])
pts2 = np.float32([[0,0],[600,0],[0,600], [600,600]])
M = cv2.getPerspectiveTransform(pts1, pts2)
dst = cv2.warpPerspective(img_rgb,M,(600,600))
plt.subplot(121),plt.imshow(img_rgb), plt.title("Original Image")
plt.subplot(122),plt.imshow(dst), plt.title("Output Image")
plt.show()
```

Output:



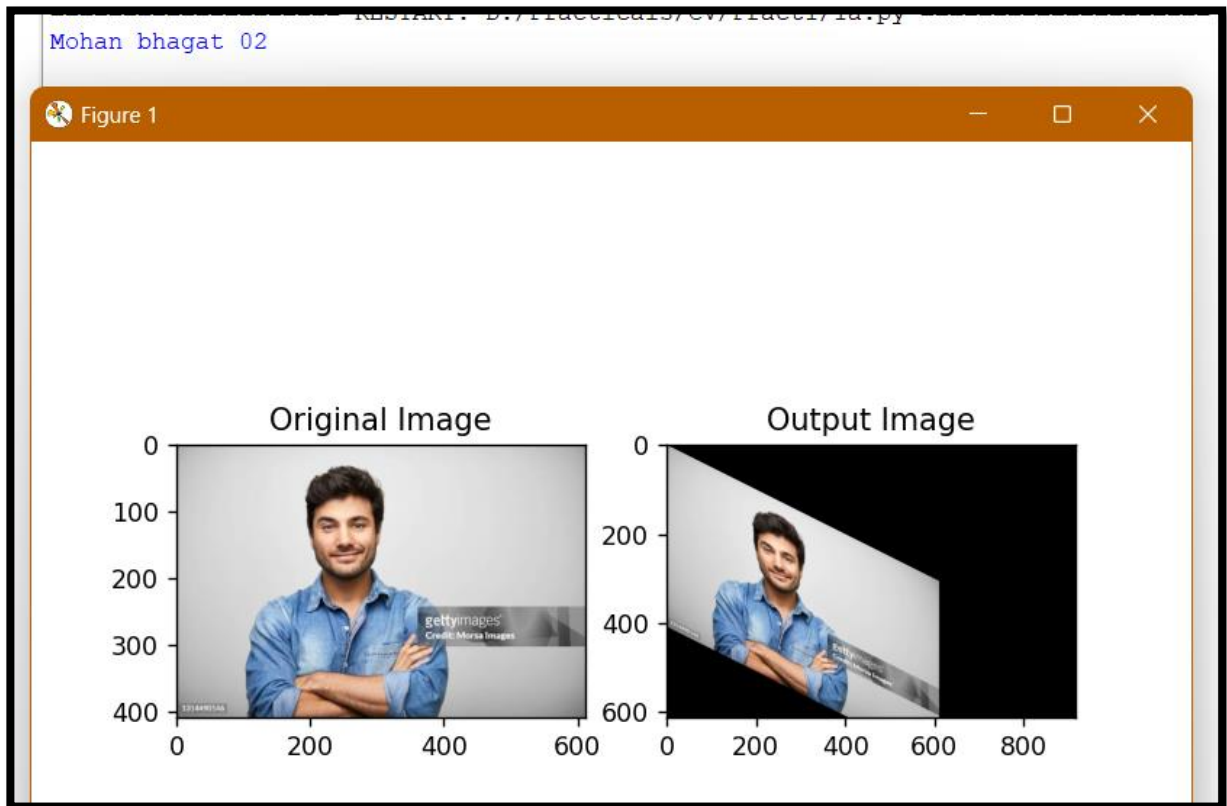
Practical – 1 (F)

Aim: Shearing X-axis.

Code:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
print("Mohan bhagat 02")
img = cv2.imread('1.jpg')
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, channels = img_rgb.shape
#For Image Shearing in Y axis
M = np.float32([[1, 0, 0], [0.5, 1, 0], [0, 0, 1]])
dst = cv2.warpPerspective(img_rgb, M,(int(cols*1.5),int(rows*1.5)))
plt.subplot(121),plt.imshow(img_rgb), plt.title("Original Image")
plt.subplot(122),plt.imshow(dst), plt.title("Output Image")
plt.show()
```

Output:



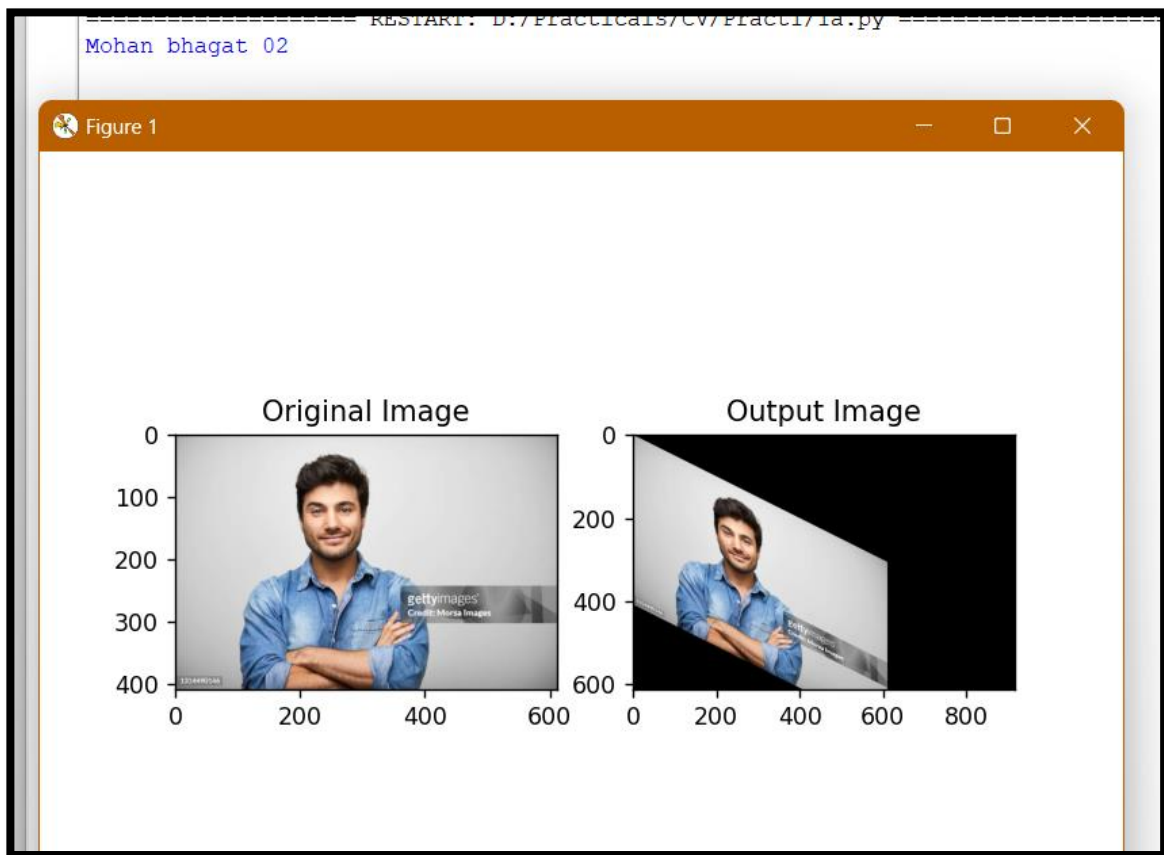
Practical – 1 (G)

Aim: Shearing Y-axis.

Code:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
print("Mohan bhagat 02")
img=cv2.imread("1.jpg")
img_rgb=cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
rows,cols,channels=img_rgb.shape
M = np.float32([[1, 0, 0], [0.5, 1, 0], [0, 0, 1]])
dst = cv2.warpPerspective(img_rgb, M, (int(cols*1.5), int(rows*1.5)))
plt.subplot(121), plt.imshow(img_rgb), plt.title('Original Image')
plt.subplot(122), plt.imshow(dst), plt.title('Output Image')
plt.show()
```

Output:



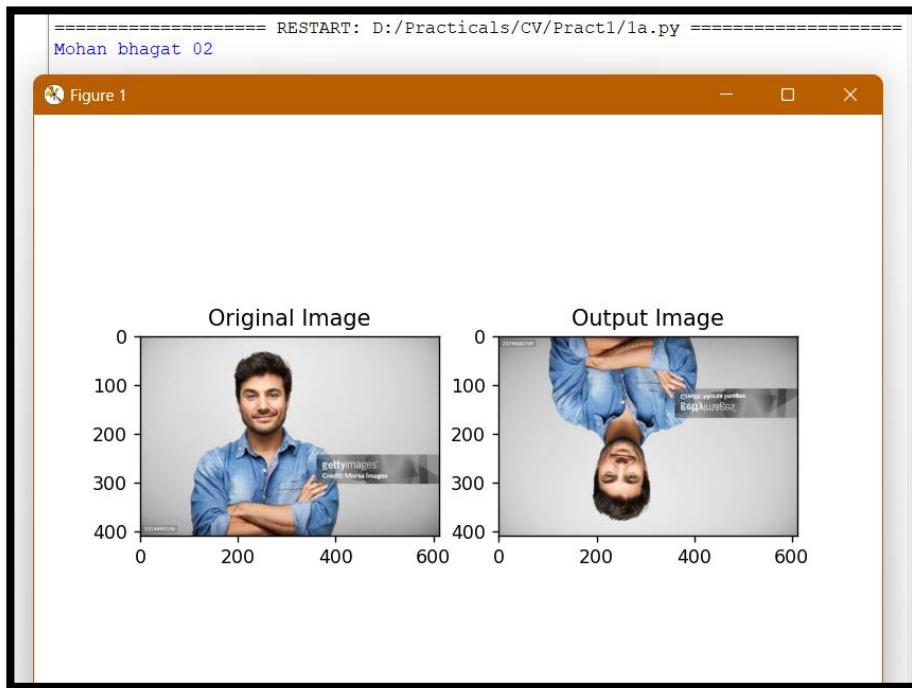
Practical – 1 (H)

Aim: Reflected Image.

Code:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
print("Mohan bhagat 02")
img = cv2.imread('1.jpg')
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, channels = img_rgb.shape
#To flip horizontally
M = np.float32([[1, 0, 0], [0, -1, rows], [0, 0, 1]])
#To flip vertically
#M = np.float32([[-1,0,cols],[0,1,0],[0,0,1]])
dst = cv2.warpPerspective(img_rgb,M,(cols,rows))
plt.subplot(121),plt.imshow(img_rgb), plt.title("Original Image")
plt.subplot(122),plt.imshow(dst), plt.title("Output Image")
plt.show()
```

Output:



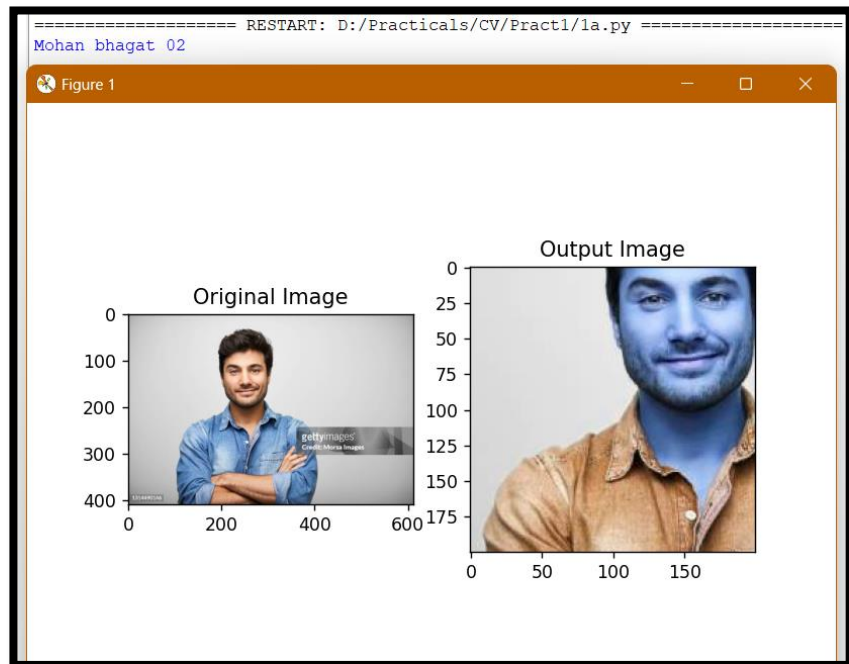
Practical – 1 (I)

Aim: Cropped Image.

Code:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
print("Mohan bhagat 02")
img = cv2.imread('1.jpg')
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
rows, cols, channels = img_rgb.shape
#We choose to get 200 pixels from 100 to 300 on both axes.
dst = img[100:300, 100:300]
plt.subplot(121),plt.imshow(img_rgb), plt.title("Original Image")
plt.subplot(122),plt.imshow(dst), plt.title("Output Image")
plt.show()
```

Output:



Practical 2

Aim: Perform Image Stitching.

Theory:

Image stitching is the process of combining multiple overlapping images to create a seamless, high-resolution output image. This technique is commonly used to create panoramic images, virtual tours, and even some medical imaging applications.

Image stitching involves several steps:

1. **Feature detection:** Identifying and extracting unique features (e.g., corners, edges) from each input image. Compute the SIFT-key points and descriptors for both the images.
2. **Feature matching:** Finding correspondences between features in the overlapping regions of the input images. Compute distances between every descriptor in one image and every descriptor in the other image. Select the top ‘m’ matches for each descriptor of an image.
3. **Homography estimation:** Estimating the transformation (e.g., rotation, scaling, translation) that aligns the input images. Run RANSAC to estimate homography
4. **Warping:** Applying the estimated transformation to the input images. Warp to align for stitching
5. **Blending:** Combining the warped images into a single seamless output image. Now stitch them together

Explanation of Code:

Firstly, we have to *find out the features matching in both the images*. These best matched features act as the basis for stitching. *We extract the key points and sift descriptors for both the images as follows:*

```
sift = cv2.SIFT_create()
# find the keypoints and descriptors with SIFT
kp1, des1 = sift.detectAndCompute(img1, None)
kp2, des2 = sift.detectAndCompute(img2, None)
```

kp1 and kp2 are keypoints, des1 and des2 are the descriptors of the respective images.

Now, the obtained descriptors in one image are to be recognized in the image too. We do that as follows:

```
bf = cv2.BFMatcher()
```

```
matches = bf.knnMatch(des1,des2, k=2)
```

The [BFMatcher\(\)](#) matches the features which are more similar. When we set parameter $k=2$, we are asking the `knnMatcher` to give out 2 best matches for each descriptor.

‘**matches**’ is a list of list, where each sub-list consists of ‘k’ objects.

Often in images, there are tremendous chances where the features may be existing in many places of the image. This may mislead us to use trivial features for our experiment. So we filter out through all the matches to obtain the best ones. So we apply ratio test using the top 2 matches obtained above. We consider a match if the ratio defined below is predominantly greater than the specified ratio.

```
# Apply ratio test
good = []
for m in matches:
    if m[0].distance < 0.5*m[1].distance:
        good.append(m)
matches = np.asarray(good)
```

It’s time to align the images now. As you know that a homography matrix is needed to perform the transformation, and the homography matrix requires at least 4 matches, we do the following.

```
if len(matches[:,0]) >= 4:
    src = np.float32([ kp1[m.queryIdx].pt for m in matches[:,0]
    ]).reshape(-1,1,2)
    dst = np.float32([ kp2[m.trainIdx].pt for m in matches[:,0]
    ]).reshape(-1,1,2)
    H, masked = cv2.findHomography(src, dst,
    cv2.RANSAC, 5.0)
    #print H
else:
    raise AssertionError("Can't find enough keypoints.")
```

And finally comes the last part, stitching of the images. Now that we found the homography for transformation, we can now proceed to warp and stitch them together:

```
dst = cv2.warpPerspective(img_,H,(img_.shape[1] +
img_.shape[1], img_.shape[0]))
plt.subplot(122),plt.imshow(dst),plt.title('Warped Image')
```

```
plt.show()
plt.figure()
dst[0:img.shape[0], 0:img.shape[1]] = img
cv2.imwrite('output.jpg',dst)
plt.imshow(dst)
plt.show()
```

We get warped image plotted using matplotlib to well visualize the warping.

Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from random import randrange
print("Mohan bhagat 02")
img_ = cv2.imread('4.jpg')
img1 = cv2.cvtColor(img_,cv2.COLOR_BGR2GRAY)
img = cv2.imread('3.JPG')
img2 = cv2.cvtColor(img_,cv2.COLOR_BGR2GRAY)
sift = cv2.SIFT_create()
#find the keypoints and descriptors with sift
kp1, des1 = sift.detectAndCompute(img1,None)
kp1, des2 = sift.detectAndCompute(img2,None)
#BFMatcher with default params
bf = cv2.BFMatcher()
matches = bf.knnMatch(des1,des2, k=2)
#print matches, Apply ratio test
good = []
for m in matches:
    if m[0].distance < 0.5*m[1].distance:
        good.append(m)
matches = np.asarray(good)
if len(matches[:,0]) >=4:
    src = np.float32([ kp1[m.queryIdx].pt for m in matches[:,0] ]).reshape(-1,1,2)
    dst = np.float32([ kp1[m.trainIdx].pt for m in matches[:,0] ]).reshape(-1,1,2)

    H, masked = cv2.findHomography(src, dst, cv2.RANSAC, 5.0)
    print(H)
else:
    raise AssertionError("Can't find enough keypoints.")
dst = cv2.warpPerspective(img_,H,(img.shape[1] + img_.shape[1], img.shape[0]))
plt.subplot(122),plt.imshow(dst),plt.title('Warped Image')
plt.show()
plt.figure()
dst[0:img.shape[0], 0:img.shape[1]] = img
cv2.imwrite('resultant_stitched_panorama.jpg',dst)
plt.imshow(dst)
plt.show()
```

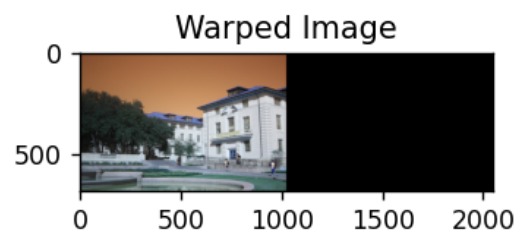
Output:

Mohan bhagat 02

```
[[ 1.00000000e+00  2.89700487e-17 -2.71535766e-12]
 [ 4.45830656e-15  1.00000000e+00 -2.77673612e-12]
 [ 9.25447385e-18 -1.44772370e-18  1.00000000e+00]]
```



Figure 1

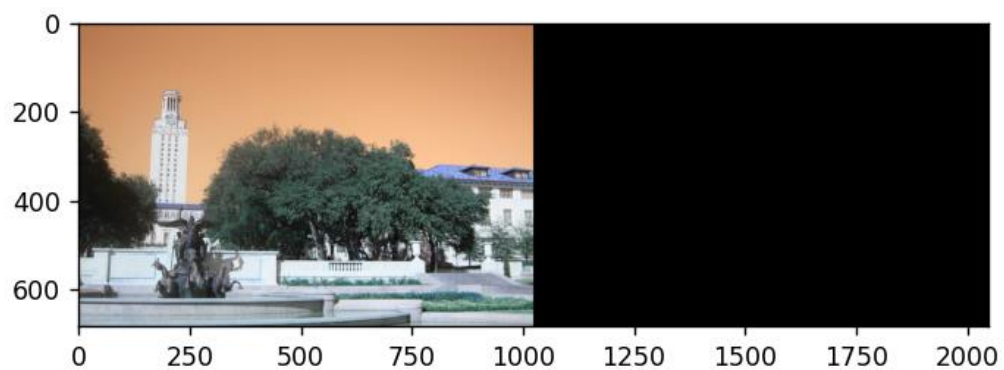


Mohan bhagat 02

```
[[ 1.00000000e+00  2.89700487e-17 -2.71535766e-12]
 [ 4.45830656e-15  1.00000000e+00 -2.77673612e-12]
 [ 9.25447385e-18 -1.44772370e-18  1.00000000e+00]]
```



Figure 1



Practical 3

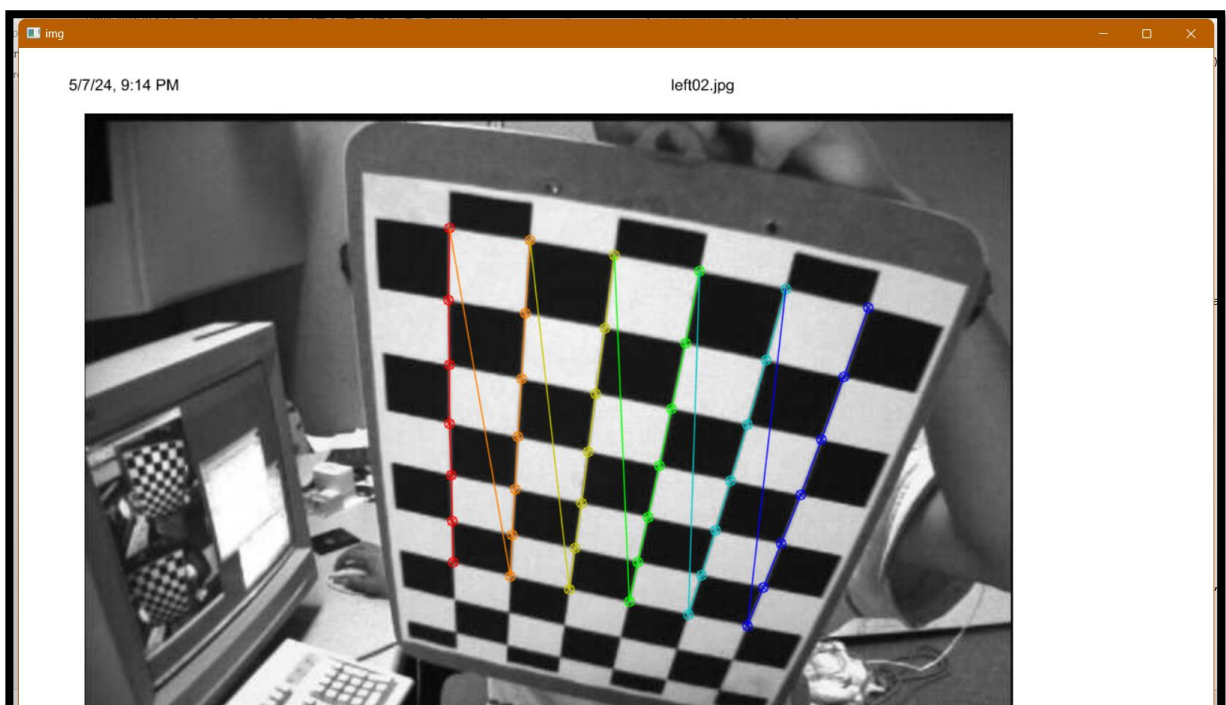
Aim: Perform Camera Calibration.

Code:

```
import numpy as np
import cv2 as cv
import glob
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30,
0.001)
print("Mohan bhagat 02")
objp = np.zeros((6*7,3), np.float32)
objp[:, :2] = np.mgrid[0:7, 0:6].T.reshape(-1,2)
objpoints = []
imgpoints = []
images = glob.glob('*.*jpg')
for fname in images:
    img = cv.imread(fname)
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    ret, corners = cv.findChessboardCorners(gray, (7,6), None)
    if ret == True:
        objpoints.append(objp)
        corners2 = cv.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)
        imgpoints.append(corners2)
        cv.drawChessboardCorners(img, (7,6), corners2, ret)
        cv.imshow('img',img)
        cv.waitKey(500)
cv.destroyAllWindows()
ret, mtx, dist, rvecs, tvecs = cv.calibrateCamera(objpoints, imgpoints, gray.shape[::-1],
None, None)
print("Camera matrix :")
print(mtx)
print("dist :")
print(dist)
print("rvecs :")
print(rvecs)
print("tvecs :")
img = cv.imread('left 8_page-0001.jpg')
h, w = img.shape[:2]
newcameramtx, roi = cv.getOptimalNewCameraMatrix(mtx, dist, (w,h), 1,(w,h))
dst = cv.undistort(img, mtx, dist, None, newcameramtx)
x, y, w, h = roi
dst = dst[y:y+h, x:x+w]
cv.imwrite('calibresult.png',dst)
```

Output:

```
Mohan bhagat 02
Camera matrix :
[[825.78194883    0.          599.54646556]
 [  0.          826.03259359  432.67675863]
 [  0.           0.           1.           ]]
dist :
[[-3.03246531e-01  1.94393960e-01  1.68216641e-03 -2.53247436e-04
 -1.51429436e-01]]
rvecs :
(array([[ -0.43094846],
        [ 0.2576368 ],
        [-3.08820456]]), array([[ 0.41712507],
        [ 0.65625708],
        [-1.33675303]]), array([[ -0.26282409],
        [-0.39215072],
        [-2.74843297]]), array([[ -0.37672468],
        [-0.17856623],
        [-3.11640627]]), array([[ -0.45803272],
        [-0.31458837],
        [-1.76158977]]), array([[ -0.29683308],
        [ 0.39008731],
        [-1.43514699]]), array([[ -0.31768401],
        [ 0.15924463],
        [-1.24156634]]), array([[ -0.45696837],
        [-0.08844467],
        [-1.33560419]]))
tvecs :
```



Practical 4

Aim: Perform the following:

- a. Face detection**
- b. Object detection**
- c. Pedestrian detection**
- d. Face recognition**

Theory:

a. Face detection

Face detection involves identifying a person's face in an image or video. This is done by analyzing the visual input to determine whether a person's facial features are present.

Since human faces are so diverse, face detection models typically need to be trained on large amounts of input data for them to be accurate. The training dataset must contain a sufficient representation of people who come from different backgrounds, genders, and cultures.

These algorithms also need to be fed many training samples comprising different lighting, angles, and orientations to make correct predictions in real-world scenarios.

These nuances make face detection a non-trivial, time-consuming task that requires hours of model training and millions of data samples.

The OpenCV package comes with pre-trained models for face detection, which means that we don't have to train an algorithm from scratch. More specifically, the library employs a machine learning approach called Haar cascade to identify objects in visual data.

Face detection approach called Haar Cascade for face detection using OpenCV and Python.

Intro to Haar Cascade Classifiers

This method was first introduced in the paper Rapid Object Detection Using a Boosted Cascade of Simple Features, written by Paul Viola and Michael Jones.

The idea behind this technique involves using a cascade of classifiers to detect different features in an image. These classifiers are then combined into one strong classifier that can accurately distinguish between samples that contain a human face from those that don't.

The Haar Cascade classifier that is built into OpenCV has already been

trained on a large dataset of human faces, so no further training is required. We just need to load the classifier from the library and use it to perform face detection on an input image.

Installing OpenCV for Python

To install the OpenCV library, simply open your command prompt or terminal window and run the following command:

```
pip install opencv-python
```

OpenCV for Face Detection in Images

We will build a detector to identify the human face in a photo. Make sure to save the picture to your working directory and rename it to `input_image` before coding along.

Step 1: Import the OpenCV Package

Now, let's import OpenCV and enter the input image path with the following lines of code:

```
import cv2  
imagePath = 'input_image.jpg'
```

Step 2: Read the Image

Then, we need to read the image with OpenCV's `imread()` function:

```
img = cv2.imread(imagePath)
```

This will load the image from the specified file path and return it in the form of a Numpy array.

Print the dimensions of this array:

```
print(img.shape)
```

```
(4000, 2667, 3)
```

Notice that this is a 3-dimensional array. The array's values represent the picture's height, width, and channels respectively. Since this is a color image, there are three channels used to depict it - blue, green, and red (BGR).

Note that while the conventional sequence used to represent images is RGB (Red, Blue, Green), the OpenCV library uses the opposite layout (Blue, Green, Red).

Step 3: Convert the Image to Grayscale

To improve computational efficiency, we first need to convert this image to

grayscale before performing face detection on it:

```
gray_image = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Step 4: Load the Classifier

Let's load the pre-trained Haar Cascade classifier that is built into OpenCV:

```
face_classifier = cv2.CascadeClassifier(  
    cv2.data.haarcascades + "haarcascade_frontalface_default.xml"  
)
```

Notice that we are using a file called `haarcascade_frontalface_default.xml`. This classifier is designed specifically for detecting frontal faces in visual input.

<https://github.com/opencv/opencv/tree/master/data/haarcascades>

OpenCV also provides other pre-trained models to detect different objects within an image - such as a person's eyes, smile, upper body, and even a vehicle's license plate.

Step 5: Perform the Face Detection

We can now perform face detection on the grayscale image using the classifier we just loaded:

```
face = face_classifier.detectMultiScale(  
    gray_image, scaleFactor=1.1, minNeighbors=5, minSize=(40, 40)  
)
```

Let's break down the methods and parameters specified in the above code:

detectMultiScale():

The `detectMultiScale()` method is used to identify faces of different sizes in the input image.

grey_image:

The first parameter in this method is called `grey_image`, which is the grayscale image we created previously.

scaleFactor:

This parameter is used to scale down the size of the input image to make it easier for the algorithm to detect larger faces. In this case, we have specified

a scale factor of 1.1, indicating that we want to reduce the image size by 10%.

minNeighbors:

The cascade classifier applies a sliding window through the image to detect faces in it. You can think of these windows as rectangles.

Initially, the classifier will capture a large number of false positives. These are eliminated using the `minNeighbors` parameter, which specifies the number of neighboring rectangles that need to be identified for an object to be considered a valid detection.

To summarize, passing a small value like 0 or 1 to this parameter would result in a high number of false positives, whereas a large number could lead to losing out on many true positives.

The trick here is to find a tradeoff that allows us to eliminate false positives while also accurately identifying true positives.

minSize:

Finally, the `minSize` parameter sets the minimum size of the object to be detected. The model will ignore faces that are smaller than the minimum size specified.

Step 6: Drawing a Bounding Box

Now that the model has detected the faces within the image, let's run the following lines of code to create a bounding box around these faces:

for (x, y, w, h) in face:

```
    cv2.rectangle(img, (x, y), (x + w, y + h), (0, 255, 0), 4)
```

The `face` variable is an array with four values: the x and y axis in which the faces were detected, and their width and height. The above code iterates over the identified faces and creates a bounding box that spans across these measurements.

The parameter `0,255,0` represents the color of the bounding box, which is green, and `4` indicates its thickness.

Step 7: Displaying the Image

To display the image with the detected faces, we first need to convert the image from the BGR format to RGB:

```
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

Use the Matplotlib library to display the image:

```
import matplotlib.pyplot as plt
```

```
plt.figure(figsize=(20,10))
```

```
plt.imshow(img_rgb)
```

```
plt.show()
```

Code:

```
import cv2
import matplotlib.pyplot as plt
print("Mohan bhagat 02")
imagePath = 'input_image_1.jpg'
img = cv2.imread(imagePath)
print(img.shape)
gary_image = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
face_classifier = cv2.CascadeClassifier(cv2.data.harcascades +
"haarcascade_frontalface_default.xml")
```

```
face = face_classifier.detectMultiScale(gary_image, scaleFactor = 1.1, minNeighbors =
5, minSize=(40,40))
```

```
for (x, y, w, h) in face:
```

```
    cv2.rectangle(img, (x, y), (x + w, y + h), (0,255,0),4)
```

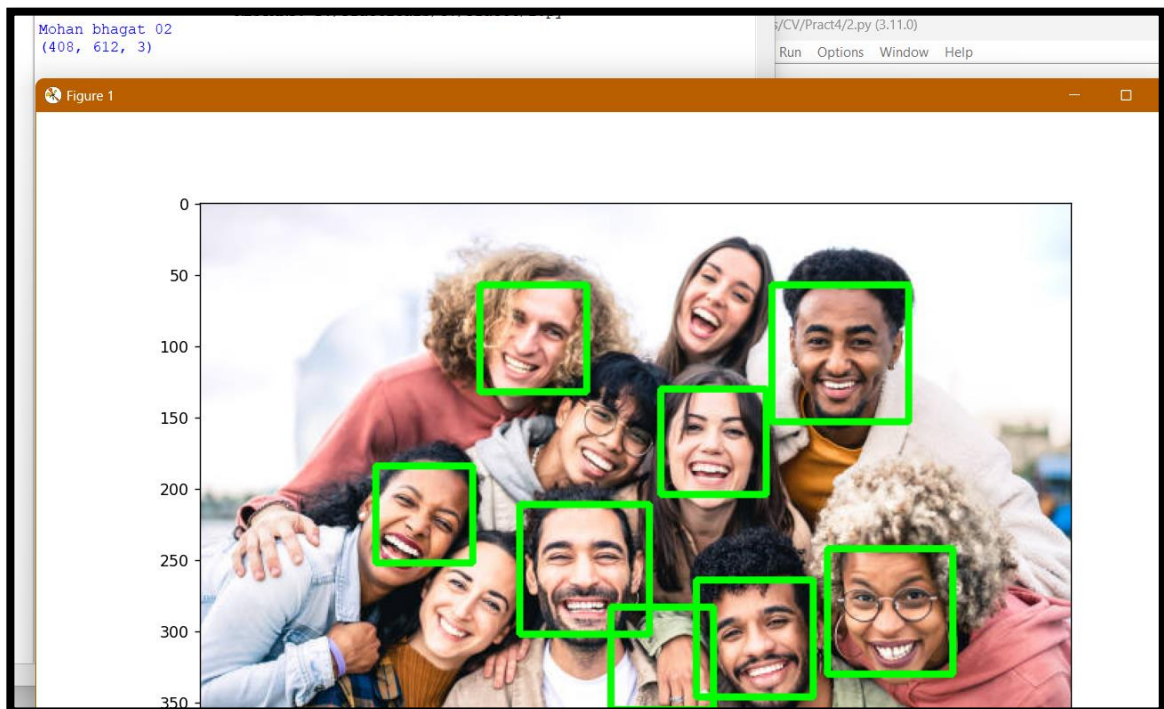
```
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

```
plt.figure(figsize=(20,10))
```

```
plt.imshow(img_rgb)
```

```
plt.show()
```

Output:



b. Object detection

Object detection is a computer technology related to computer vision and image processing that deals with detecting instances of semantic objects of a certain class (such as humans, buildings, or cars) in digital images and videos.

Haar cascade:

Basically, the Haar cascade technique is an approach based on machine learning where we use a lot of positive and negative images to train the classifier to classify between the images. Haar cascade classifiers are considered as the effective way to do object detection with the OpenCV library.

Positive images: These are the images that contain the objects which we want to be identified from the classifier.

Negative Images: These are the images that do not contain any object that we want to be detected by the classifier, and these can be images of everything else.

Requirements for object detection with Python OpenCV:

- 1. Install OpenCV-Python Library**
- 2. Install matplotlib library**
- 3. An Image with Stop Sign.**
- 4. An xml file “Stop_data.xml” to detect stop sign board.**

Code:

```
import cv2
from matplotlib import pyplot as plt
imaging = cv2.imread("image.jpg")
print("Mohan Bhagat 02")
imaging_gray = cv2.cvtColor(imaging, cv2.COLOR_BGR2GRAY)
imaging_rgb = cv2.cvtColor(imaging, cv2.COLOR_BGR2RGB)
xml_data = cv2.CascadeClassifier('stop_data.xml')
detecting = xml_data.detectMultiScale(imaging_gray, minSize=(30, 30))
amountDetecting = len(detecting)
if amountDetecting != 0:
    for (a, b, width, height) in detecting:
        cv2.rectangle(imaging_rgb, (a, b), # Highlighting detected object with rectangle
                      (a + width, b + height),
                      (0, 255, 0), 9)
plt.imshow(imaging_rgb)
plt.show()
```

Output:



Line Detection

So, how does the Hough Transform algorithm work? Let's dive into the details.

Each point in the image is transformed into a line in the parameter space. This conversion allows us to represent lines in a different coordinate system, making line detection more feasible. Think of it as a translation from the image domain to a space where lines become easier to analyze.

The **Hough Transform algorithm** is based on the following assumptions:

- Lines in an image can be represented by their equation in polar coordinates, which is given by: $r = x\cos(\theta) + y\sin(\theta)$, where (r, θ) are the parameters of the line, and (x, y) are the coordinates of a point on the line.
- A point in an image corresponds to a sinusoidal curve in the parameter space.
- The intersection of curves in the parameter space corresponds to a line in the image.

By considering these assumptions, we can establish that a point in an image corresponds to a sinusoidal curve in the parameter space. As we detect more points, these curves intersect and accumulate at certain locations. These intersections indicate the presence of lines in the image.

Line Detection Algorithm using OpenCV Python

Now that we have a fundamental grasp of the Hough Transform and Probabilistic Hough Transform algorithm, let's explore their implementation using OpenCV and Python.

To identify lines in an image using OpenCV in Python, we can proceed with the following steps:

1. Import Required Libraries.
2. Load the image.
3. Convert the image to grayscale.
4. Apply Canny Edge Detection to get the edges.
5. Apply Hough Line Transform to detect the lines.
6. Draw the detected lines on the image.

Convert the Image to Grayscale

The Hough Transform algorithm works on grayscale images, so we need to convert the image to grayscale using the `cv2.cvtColor()` function.

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Apply Edge Detection

Next, we need to apply an `edge detection` algorithm to the grayscale image. In this example, we will use the Canny edge detection algorithm, which is a popular edge detection algorithm that is based on the gradient of the image.

```
edges = cv2.Canny(gray, 50, 150, apertureSize=3)
```

Apply the Hough Transform Algorithm

OpenCV provides several algorithms for line detection, including **HoughLines** and **HoughLinesP**.

HoughLines is a standard Hough transform algorithm that returns an array of lines in the image. **HoughLinesP** is a probabilistic Hough transform algorithm that is faster and more accurate than HoughLines. It returns an array of line segments instead of full lines.

Now, we can apply the Hough Transform algorithm using the `cv2.HoughLines()` or `cv2.HoughLinesP` function.

The function takes the edge image as input and returns a list of lines detected in the image. The parameters of the function are explained below:

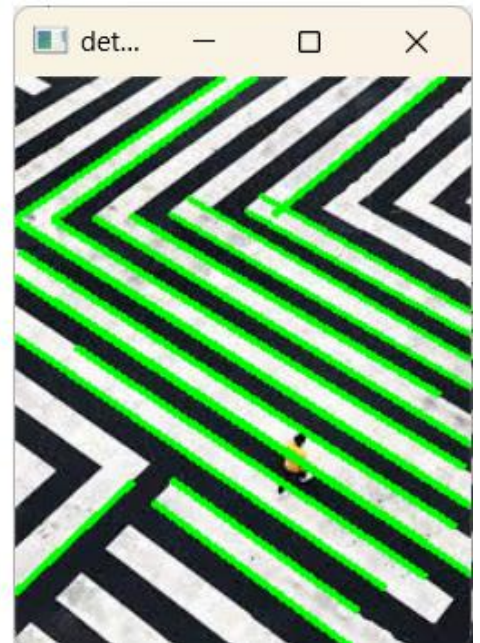
- **rho**: The distance resolution in pixels of the Hough grid.
- **theta**: The angular resolution in radians of the Hough grid.
- **threshold**: The minimum number of votes (intersections in the Hough grid) required to detect a line.
- **minLineLength**: The minimum length of a line. Lines shorter than this will be discarded.
- **maxLineGap**: The maximum allowed gap between two points on the same line.

- **Using HoughLinesP function:**
- `lines = cv2.HoughLinesP(edges, rho=1, theta=np.pi/180, threshold=100, minLineLength=100, maxLineGap=10)`

Code:

```
import cv2
import numpy as np
image = cv2.imread('1.jpg')
cv2.imshow('Original', image)
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
edges = cv2.Canny(gray, 50, 150, apertureSize=3)
lines_list = []
lines = cv2.HoughLinesP(
    edges,
    rho=1,
    theta= np.pi/180,
    threshold=100,
    minLineLength =3,
    maxLineGap=5
)
for points in lines:
    x1,y1,x2,y2=points [0]
    cv2.line(image, (x1,y1), (x2, y2), (0,255,0),2)
    lines_list.append([ (x1,y1), (x2,y2)])
cv2.imshow('detectedLines.png', image)
```

Output:



Hough transform

- Finds circles in a grayscale image using the Hough transform.
- The function finds circles in a grayscale image using a modification of the Hough transform.

```
void cv::HoughCircles ( InputArray  image,  
                        OutputArray circles,  
                        int          method,  
                        double       dp,  
                        double       minDist,  
                        double       param1 = 100 ,  
                        double       param2 = 100 ,  
                        int          minRadius = 0 ,  
                        int          maxRadius = 0  
                        )
```

- Usually the function detects the centers of circles well. However, it may fail to find correct radii.
- You can assist to the function by specifying the radius range (minRadius and maxRadius) if you know it.

- Or, in the case of `HOUGH_GRADIENT` method you may set `maxRadius` to a negative number to return centers only without radius search, and find the correct radius using an additional procedure.
- It also helps to smooth image a bit unless it's already soft. For example, `GaussianBlur()` with 7x7 kernel and 1.5x1.5 sigma or similar blurring may help.

Parameters

image	8-bit, single-channel, grayscale input image.
circles	Output vector of found circles. Each vector is encoded as 3 or 4 element floating-point vector $(x, y, radius)$ or $(x, y, radius, votes)$.
method	Detection method, see HoughModes . The available methods are <code>HOUGH_GRADIENT</code> and <code>HOUGH_GRADIENT_ALT</code> .
dp	Inverse ratio of the accumulator resolution to the image resolution. For example, if <code>dp=1</code> , the accumulator has the same resolution as the input image. If <code>dp=2</code> , the accumulator has half as big width and height. For <code>HOUGH_GRADIENT_ALT</code> the recommended value is <code>dp=1.5</code> , unless some small very circles need to be detected.
minDist	Minimum distance between the centers of the detected circles. If the parameter is too small, multiple neighbor circles may be falsely detected in addition to a true one. If it is too large, some circles may be missed.
param1	First method-specific parameter. In case of <code>HOUGH_GRADIENT</code> and <code>HOUGH_GRADIENT_ALT</code> , it is the higher threshold of the two passed to the Canny edge detector (the lower one is twice smaller). Note that <code>HOUGH_GRADIENT_ALT</code> uses Scharr algorithm to compute image derivatives, so the threshold value should normally be higher, such as 300 or normally exposed and contrasty images.
param2	Second method-specific parameter. In case of <code>HOUGH_GRADIENT</code> , it is the accumulator threshold for the circle centers at the detection stage. The smaller it is, the more false circles may be detected. Circles, corresponding to the larger accumulator values, will be returned first. In the case of <code>HOUGH_GRADIENT_ALT</code> algorithm, this is the circle "perfectness" measure. The closer it is to 1, the better shaped circles algorithm selects. In most cases 0.9 should be fine. If you want get better detection of small circles, you may decrease it to 0.85, 0.8 or even less. But then also try to limit the search range [<code>minRadius</code> , <code>maxRadius</code>] to avoid many false circles.
minRadius	Minimum circle radius.
maxRadius	Maximum circle radius. If <code><= 0</code> , uses the maximum image dimension. If <code>< 0</code> , <code>HOUGH_GRADIENT</code> returns centers without finding the radius. <code>HOUGH_GRADIENT_ALT</code> always computes circle radiuses.

What is Hough accumulator?

- The Hough transform algorithm uses an array, called an accumulator, to detect the existence of a line $y = mx + b$. The dimension of the accumulator is equal to the number of unknown parameters of the Hough transform problem.

Overview of `np.around()`

Basically, `np.around()` is NumPy's function to round floats (decimal numbers) or integers. It takes three arguments:

1. The first mandatory argument is the number you want to round
2. `decimals` - optional argument, the number of decimal places you want to round to
3. `out` - optional argument, where to output

By default `np.around()` will round the given number to the nearest whole number and return the rounded value:

```
>>> np.around(5)
5

>>> np.around(5.11212)
5.0

>>> np.around(9.11212)
9.0

>>> np.around(9.5)
10.0
```

OpenCV-Python is a library of Python bindings designed to solve computer vision problems. `cv2.circle()` method is used to draw a circle on any image. The syntax of `cv2.circle()` method is:

Syntax:

```
cv2.circle(image, center_coordinates, radius, color, thickness)
```

Parameters:

- **image:** It is the image on which the circle is to be drawn.
- **center_coordinates:** It is the center coordinates of the circle. The coordinates are represented as tuples of two values i.e. (X coordinate value, Y coordinate value).
- **radius:** It is the radius of the circle.
- **color:** It is the color of the borderline of a circle to be drawn. For **BGR**, we pass a tuple. eg: (255, 0, 0) for blue color.
- **thickness:** It is the thickness of the circle border line in **px**. Thickness of **-1 px** will fill the circle shape by the specified color.

Return Value: It returns an image.

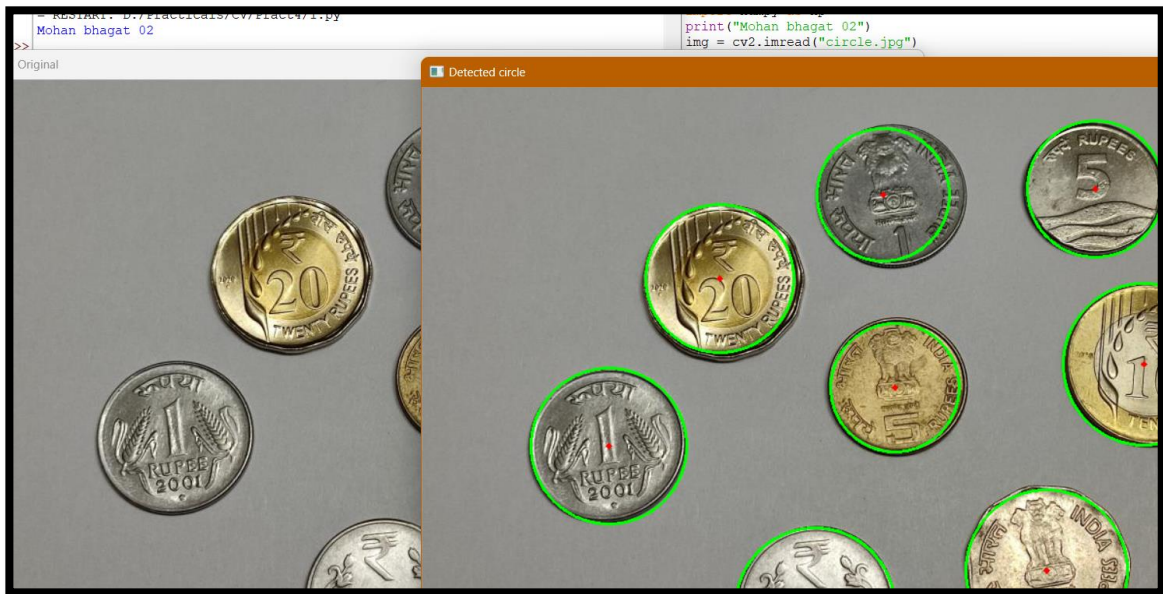
Code:

```
import cv2
import numpy as np
print("Mohan bhagat 02")
img = cv2.imread("circle.jpg")
cv2.imshow("Original", img)

gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
gray_blurred = cv2.blur(gray,(3,3))
detected_circles = cv2.HoughCircles(gray_blurred,
cv2.HOUGH_GRADIENT,1,20,param1 = 200, param2 = 80, minRadius=2,
maxRadius=90)

if detected_circles is not None:
    detected_circles = np.uint16(np.around(detected_circles))
    for pt in detected_circles[0,:]:
        a,b,r = pt[0],pt[1],pt[2]
        cv2.circle(img,(a,b),r, (0,255,0),2)
        cv2.circle(img,(a,b),1,(0,0,255),3)
    cv2.imshow("Detected circle", img)
```

Output:



Practical No. 4C

Aim: Perform the following:

- a. Face detection
- b. Object detection
- c. Pedestrian detection
- d. Face recognition

Theory:

c. Pedestrian detection

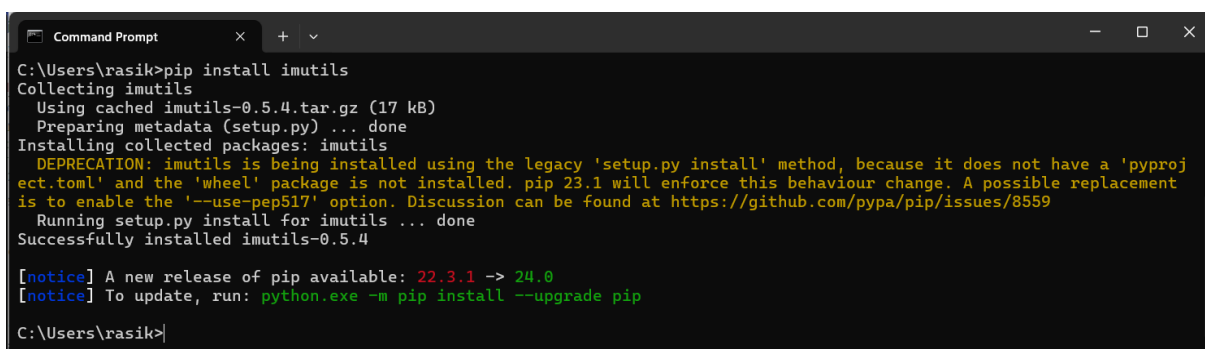
Pedestrian detection is a very important area of research because it can enhance the functionality of a pedestrian protection system in Self Driving Cars. We can extract features like head, two arms, two legs, etc, from an image of a human body and pass them to train a machine learning model. After training, the model can be used to detect and track humans in images and video streams. However, OpenCV has a built-in method to detect pedestrians. It has a pre-trained HOG(Histogram of Oriented Gradients) + Linear SVM model to detect pedestrians in images and video streams.

Histogram of Oriented Gradients

This algorithm checks directly surrounding pixels of every single pixel. The goal is to check how darker is the current pixel compared to the surrounding pixels. The algorithm draws arrows showing the direction of the image getting darker. It repeats the process for each and every pixel in the image. At last, every pixel would be replaced by an arrow, these arrows are called Gradients. These gradients show the flow of light from light to dark. By using these gradients algorithms perform further analysis.

Requirements

1. opencv-python
2. imutils



```
C:\Users\rasik>pip install imutils
Collecting imutils
  Using cached imutils-0.5.4.tar.gz (17 kB)
  Preparing metadata (setup.py) ... done
Installing collected packages: imutils
  DEPRECATION: imutils is being installed using the legacy 'setup.py install' method, because it does not have a 'pyproject.toml' and the 'wheel' package is not installed. pip 23.1 will enforce this behaviour change. A possible replacement is to enable the '--use-pep517' option. Discussion can be found at https://github.com/pypa/pip/issues/8559
  Running setup.py install for imutils ... done
Successfully installed imutils-0.5.4

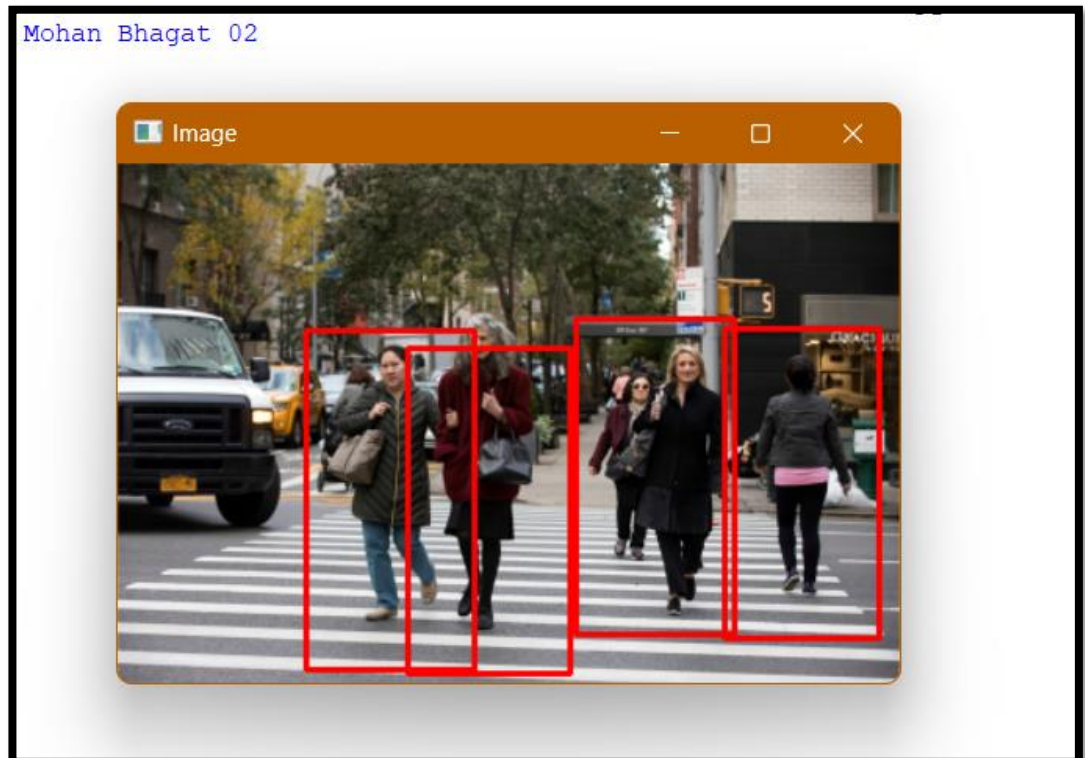
[notice] A new release of pip available: 22.3.1 -> 24.0
[notice] To update, run: python.exe -m pip install --upgrade pip

C:\Users\rasik>
```

Code:

```
import cv2
import imutils
print("Mohan Bhagat 02")
hog = cv2.HOGDescriptor()
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())
image = cv2.imread('image.jpg')
image = imutils.resize(image,width=min(400,image.shape[1]))
(regions, _) = hog.detectMultiScale(image,winStride=(4,4),padding=(4,4),scale=1.05)
for (x, y, w, h) in regions:
    cv2.rectangle(image,(x,y),(x + w, y + h),(0, 0, 255),2)
cv2.imshow("Image",image)
```

Output:



Practical No. 4D

Aim: Perform the following Face Recognition.

Theory:

Face recognition is different from face detection. In face detection, we had only detected the location of human faces, and we recognized the identity of faces in the face recognition task. In this article, we are going to build a face recognition system using python with the help of face recognition library.

There are many algorithms available in the market for face recognition. This broad computer vision challenge is detecting faces from videos and pictures. Many applications can be built on top of recognition systems. Many big companies are adopting recognition systems for their security and authentication purposes.

Use Cases of Recognition Systems

Face recognition systems are widely used in the modern era, and many new innovative systems are built on top of recognition systems.

There are a few used cases :

- Finding Missing Person
- Identifying accounts on social media
- Recognizing Drivers in Cars
- School Attendance System

Several methods and algorithms implement facial recognition systems depending on the performance and accuracy.

Traditional Face Recognition Algorithm

Traditional face recognition algorithms don't meet modern-day's facial recognition standards. They were designed to recognize faces using old conventional algorithms.

OpenCV provides some traditional facial Recognition Algorithms.

- Eigenfaces
- Scale Invariant Feature Transform (SIFT)
- Fisher faces MSC IT Part 1 Sem 2 Roll No.: 07 Computer Vision
- Local Binary Patterns Histograms (LBPH)

These methods differ in the way they extract image information and match input and output images. LBPH algorithm is a simple yet very efficient method still in use but it's slow compared to modern days algorithms.

Deep Learning For Face Recognition

There are various deep learning-based facial recognition algorithms available.

- DeepFace
- DeepID series of systems,
- FaceNet
- VGGFace

Generally, face recognizers that are based on landmarks take face images and try to find essential feature points such as eyebrows, corners of the mouth, eyes, nose, lips, etc.

There are more than 60 points. Steps Involved in Face Recognition

1. **Face Detection:** Locate the face, note the coordinates of each face locate,d and draw a bounding box around every faces.
2. **Face Alignments.** Normalize the faces in order to attain fast training.
3. **Feature Extraction.** Local feature extraction from facial pictures for training, this step is performed differently by different algorithms.
4. **Face Recognition.** Match the input face with one or more known faces in our dataset.

Code:

```
import cv2
import numpy as np
import face_recognition

print("Mohan Bhagat 02")
img_bgr = face_recognition.load_image_file('1.jpg')
img_rgb = cv2.cvtColor(img_bgr,cv2.COLOR_BGR2RGB)
cv2.imshow('bgr', img_bgr)
cv2.imshow('rgb', img_rgb)

img_tony=face_recognition.load_image_file('1.jpg')
img_tony_rgb = cv2.cvtColor(img_tony,cv2.COLOR_BGR2RGB)
#----- Detecting Face -----
face = face_recognition.face_locations(img_tony_rgb)[0]
copy = img_tony_rgb.copy()
# ----- Drawing bounding boxes around Faces-----
cv2.rectangle(copy, (face[3], face[0]),(face[1], face[2]), (255,0,255), 2)
cv2.imshow('copy', copy)
cv2.imshow('tony',img_tony_rgb)
cv2.waitKey(0)

img_tony = face_recognition.load_image_file('1.jpg')
img_tony = cv2.cvtColor(img_tony,cv2.COLOR_BGR2RGB)

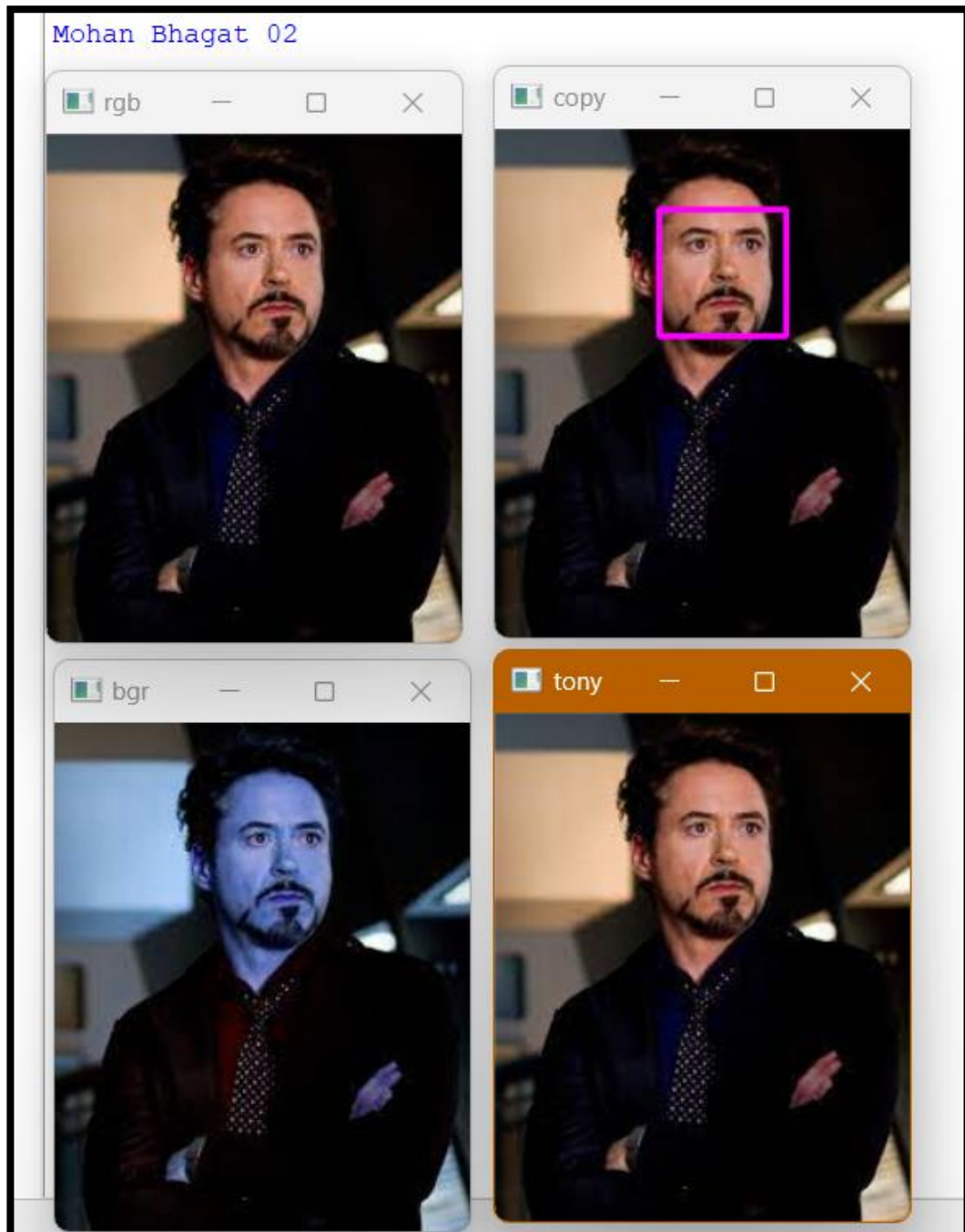
#-----to find the face location
face = face_recognition.face_locations(img_tony)[0]
```

```

#--Converting image into encodings
train_encode = face_recognition.face_encodings(img_tony)[0]

#---- lets test an image
test = face_recognition.load_image_file('2.jpg')
test = cv2.cvtColor(test, cv2.COLOR_BGR2RGB)
test_encode = face_recognition.face_encodings(test)[0]
print(face_recognition.compare_faces([train_encode],test_encode))
cv2.rectangle(img_tony, (face[3], face[0]),(face[1], face[2]), (255,0,255), 1)
cv2.imshow('img_tony', img_tony)
cv2.waitKey(0)
Output:

```



Practical 5

Aim: Implement object detection and tracking from video.

Theory:

Object detection is the detection on every single frame and frame after frame.

Object tracking does frame-by-frame tracking but keeps the history of where the object is at a time after time.

1. Importing Libraries and Modules:

Import cv2:

Imports the OpenCV library used for computer vision tasks.

From tracker import *:

Imports all functions and classes from a tracker module, which likely

Contains the implementation of the EuclideanDistTracker.

2. Creating Objects and Initializing Video Capture:

tracker = EuclideanDistTracker():

Instantiates the Euclidean Distance Tracker.

cap = cv2.VideoCapture("highway.mp4"):

Initializes video capture with the video file "highway.mp4".

3. Object Detection Initialization:

object_detector =

cv2.createBackgroundSubtractorMOG2(history=100,

varThreshold=40):

Initializes a background subtractor with MOG2 method to differentiate

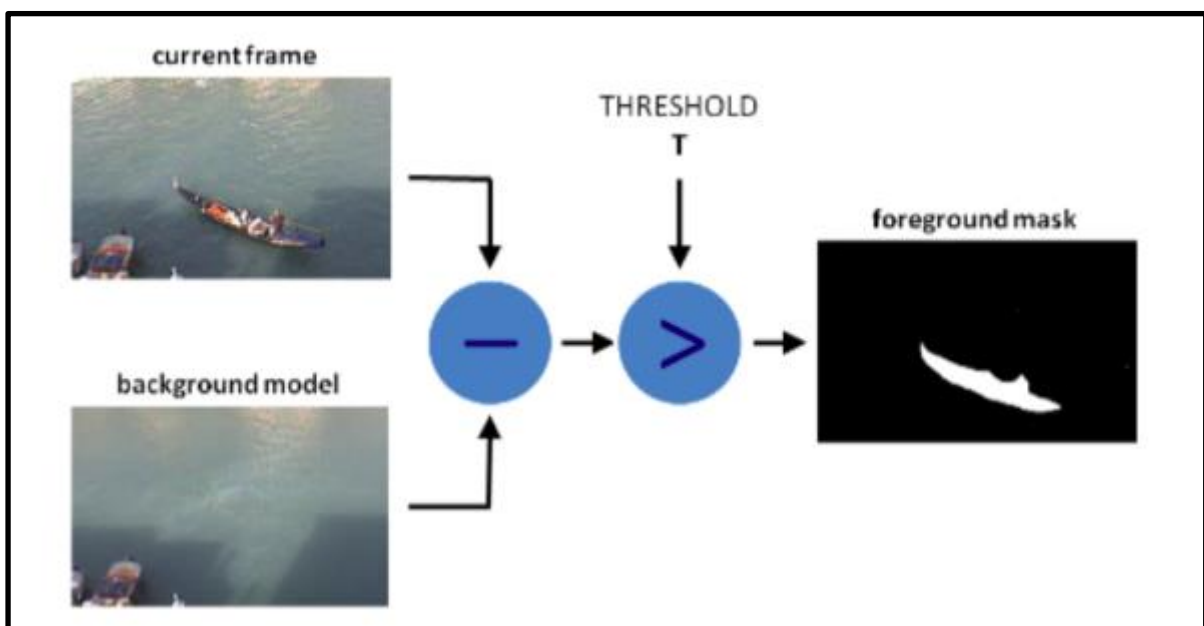
between foreground (moving objects) and the background.

Background subtraction (BS) is a common and widely used technique for

generating a foreground mask (namely, a binary image containing the pixels belonging to moving objects in the scene) by using static cameras.

As the name suggests, BS calculates the foreground mask performing a

subtraction between the current frame and a background model, containing the static part of the scene or, more in general, everything that can be considered as background given the characteristics of the observed Scene.



4. Processing Video Frames: The while True loop starts an infinite loop to process video frames until manually stopped.

`ret, frame = cap.read()`: Reads the next frame from the video.

`height, width, _ = frame.shape`: Retrieves the dimensions of the frame.

5. Defining Region of Interest (ROI):

`roi = frame[340: 720, 500: 800]`: Defines a specific area in the video frame

to focus the object detection on. This reduces computation and ignores

irrelevant areas.

6. Object Detection:

`mask = object_detector.apply(roi):` Applies the background subtractor to

the ROI to get the foreground mask.

`_, mask = cv2.threshold(mask, 254, 255, cv2.THRESH_BINARY):`

Applies a threshold to the mask to make it binary, which helps in identifying distinct

objects.
`contours, _ = cv2.findContours(mask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE):`

Finds the contours of the detected objects in the binary mask.

7. Filtering and Storing Detections:

It iterates through each contour,

calculates its area, and if the area is larger than a threshold (100), it

calculates a bounding box for the object. These bounding boxes ([x, y, w, h]) are added to the detections list representing detected objects.

8. Object Tracking: `boxes_ids = tracker.update(detections):`

The tracker updates with the current frame's detections and returns the tracked objects

with their IDs. The loop then iterates through these tracked objects, drawing their ID and bounding box on the ROI.

9. Displaying Results: `cv2.imshow():`

Displays the ROI, the original frame, and the binary mask in separate windows.
`key = cv2.waitKey(30):` Waits

for a key press for 30 ms and breaks the loop if the 'Esc' key is pressed.

10. Cleanup: `cap.release()` and `cv2.destroyAllWindows():`

Releases the video capture and closes all OpenCV windows.

In summary, this script is a complete system for object detection and tracking in a video. It uses MOG2 for background subtraction to detect moving objects and tracks them using a Euclidean Distance Tracker. It processes and visualizes the results in real-time until the user exits by pressing the 'Esc'

Code:

`import cv2`

```

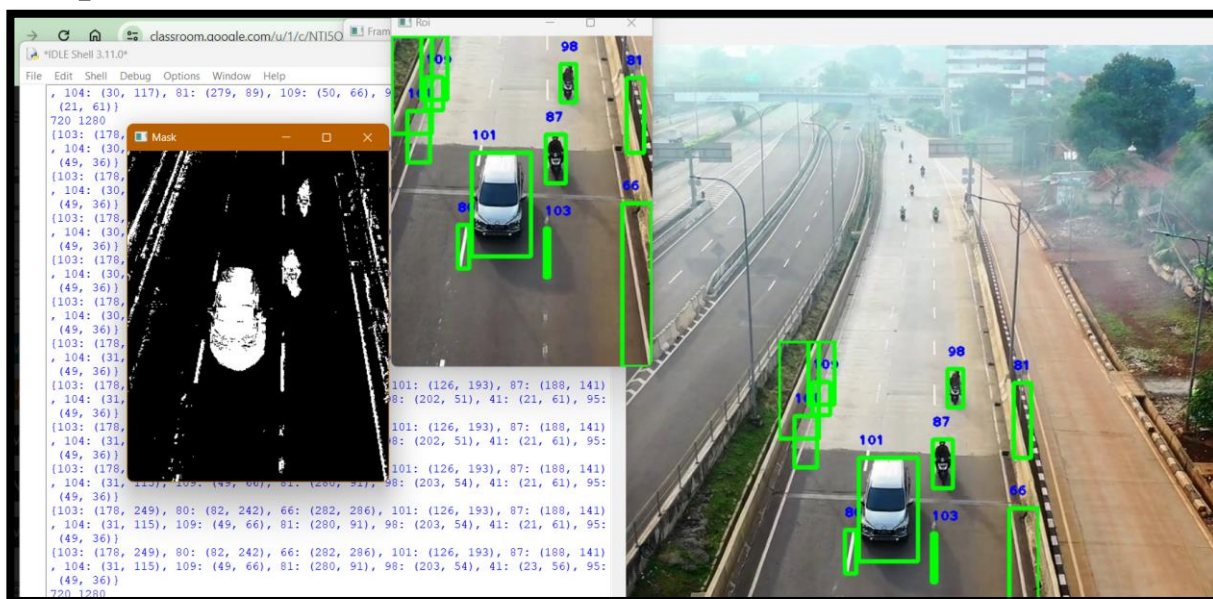
from tracker import *
print("Mohan bhagat 02")
tracker = EuclideanDistTracker()
cap = cv2.VideoCapture("highway.mp4")
object_detector = cv2.createBackgroundSubtractorMOG2(history = 100, varThreshold =
40)
while True:
    ret, frame = cap.read()
    height, width, _ = frame.shape
    print(height, width)
    roi = frame[340: 720, 500:800]
    mask = object_detector.apply(roi)
    _, mask = cv2.threshold(mask, 254, 255, cv2.THRESH_BINARY)
    contours, _ = cv2.findContours(mask, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)
    detections = []
    for cnt in contours:
        area = cv2.contourArea(cnt)
        if area > 100:
            x,y,w,h = cv2.boundingRect(cnt)
            cv2.rectangle(roi,(x, y), (x + w, y + h), (0,255,0), 2)
            detections.append([x, y, w, h])
    boxes_ids = tracker.update(detections)
    for box_id in boxes_ids:
        x ,y, w, h, id = box_id
        cv2.putText(roi, str(id), (x ,y -15), cv2.FONT_HERSHEY_PLAIN, 1, (255, 0, 0),
2)
        cv2.rectangle(roi, (x,y), (x+w , y+h), (0,255,0), 3)
    cv2.imshow("Roi", roi)
    cv2.imshow("Frame", frame)
    cv2.imshow("Mask", mask)

    key = cv2.waitKey(30)
    if key == 27:
        break

cap.release()
cv2.destroyAllWindows()

```

Output:



b)

Aim: Implement object detection and tracking from video.

B] Count number of Faces using Python

We will use image processing to detect and count the number of faces. We are

not supposed to get all the features of the face. Instead, the objective is to obtain

the bounding box through some methods i.e. coordinates of the face in the image, depending on different areas covered by the number of the coordinates,

number faces that will be computed.

Required libraries:

OpenCV library in python is a computer vision library, mostly used for image processing, video processing, and analysis, facial recognition and detection, etc.

Dlib library in python contains the pre-trained facial landmark detector, that is used to detect the (x, y) coordinates that map to facial structures on the face.

Numpy is a general-purpose array-processing package. It provides a high-performance multidimensional array object and tools for working with these arrays

Code:

```
import cv2
import numpy as np
import dlib
print("Mohan bhagat 02")
cap = cv2.VideoCapture ("Faces1.mp4")
detector=dlib.get_frontal_face_detector ()
while True:
    ret, frame = cap.read()
    frame = cv2.flip (frame, 1)
    gray = cv2.cvtColor (frame, cv2.COLOR_BGR2GRAY)
    faces =detector (gray)
    i = 0
    for face in faces:
        # Get the coordinates of faces
        x, y = face. left (), face.top()
        x1, y1 = face.right (), face.bottom ()
        cv2.rectangle (frame, (x, y), (x1, y1), (0, 255, 0), 2)
        i=i+1
        cv2.putText (frame, 'face num' + str(i), (x-10, y-
10),cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255), 3)
        print (face, i)
    frame = cv2.resize (frame, (1500, 1000))
    cv2.imshow('frame', frame)
    if cv2.waitKey (1) & 0xFF== ord('q'):
        break
# Release the capture and destroy the windows
cap.release()
cv2.destroyAllWindows ()
```

Output:

```

Mohan bhagat 02
[(3052, 782) (3267, 997)] 1
[(1881, 591) (2096, 806)] 2
[(1045, 758) (1260, 973)] 3
[(2407, 711) (2622, 926)] 4
[(3598, 791) (3777, 970)] 5
[(509, 595) (767, 853)] 6
[(1881, 591) (2096, 806)] 1
[(3052, 782) (3267, 997)] 2
[(1045, 758) (1260, 973)] 3
[(2431, 711) (2646, 926)] 4
[(3598, 791) (3777, 970)] 5
[(509, 595) (767, 853)] 6
[(1881, 591) (2096, 806)] 1
[(3032, 738) (3290, 996)] 2
[(2431, 711) (2646, 926)] 3
[(1045, 758) (1260, 973)] 4
[(509, 595) (767, 853)] 5
[(3598, 791) (3777, 970)] 6
[(1881, 591) (2096, 806)] 1
[(2431, 711) (2646, 926)] 2
[(1045, 758) (1260, 973)] 3
[(3076, 782) (3291, 997)] 4
[(509, 595) (767, 853)] 5
[(3032, 738) (3290, 996)] 2

```



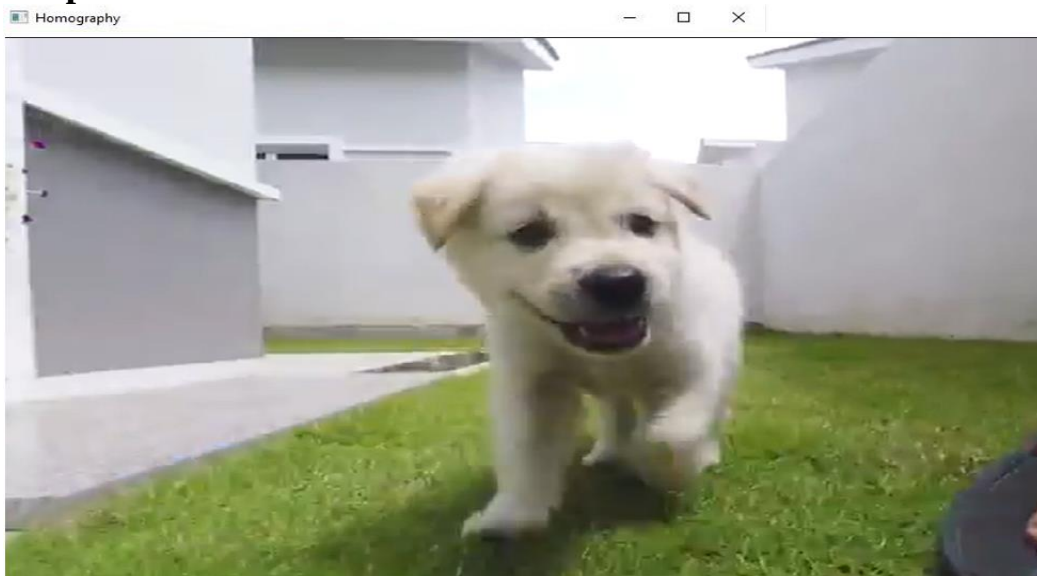
c)

Aim: Implement object detection and tracking from video. (Object Tracking using Homography).

Code:

```
import cv2
import numpy as np
img = cv2.imread("dog.png", cv2.IMREAD_GRAYSCALE)
cap = cv2.VideoCapture("dog.mp4")
sift = cv2.SIFT_create()
kp_image, desc_image = sift.detectAndCompute(img, None)
index_params = dict( algorithm = 0, trees = 5)
search_params = dict()
flann = cv2.FlannBasedMatcher (index_params, search_params)
_, frame = cap.read()
grayframe = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
kp_grayframe, desc_grayframe = sift.detectAndCompute(grayframe, None)
matches = flann.knnMatch(desc_image, desc_grayframe, k = 2)
good_points = []
for m,n in matches:
    if m.distance < 0.6 *n.distance:
        good_points.append(m)
query_pts = np.float32([kp_image[m.queryIdx].pt for m in good_points]).reshape(-1, 1, 2)
train_pts = np.float32([kp_grayframe[m.trainIdx].pt for m in good_points]).reshape(-1, 1, 2)
matrix, mask = cv2.findHomography(query_pts, train_pts, cv2.RANSAC, 5.0)
matches_mask = mask.ravel().tolist()
h, w = img.shape
pts = np.float32([[0,0],[0,h],[w,h], [w,0]]).reshape(-1, 1, 2)
dst = cv2.perspectiveTransform(pts, matrix)
homography = cv2.polylines(frame, [np.int32(dst)], True, (255,0,0),3)
cv2.imshow("Homography",homography)
```

Output:



Practical 6

Aim: Perform Colorization.

Theory:

We'll create a program to convert a black & white image i.e grayscale image to a colour image. We're going to use the **Caffe colourization model** for this program. And you should be familiar with basic OpenCV functions and uses like reading an image or how to load a pre-trained model using dnn module etc.

The procedure that we'll follow to implement the program:

Steps:

1. Load the model and the **convolution/kernel points**
2. Read and preprocess the image
3. Generate model predictions using the **L channel** from our input image
4. Use the output -> **ab channel** to create a resulting image

What is the L channel and ab channel? Basically like **RGB** colour space, there is something similar, known as **Lab colour space**. And this is the basis on which our program is based. Let's discuss what it is briefly:

What is Lab Colour Space?

Like RGB, lab colour has 3 channels L, a, and b. But here instead of pixel values, these have different significances i.e :

- **L-channel:** light intensity
- **a channel:** green-red encoding
- **b channel:** blue-red encoding

And In our program, we'll use the L channel of our image as input to our model to predict ab channel values and then rejoin it with the L channel to generate our final image.

Automatic colorization of photos using deep neural networks is a technology that can add color to black and white photos without the need for manual coloring. This technology uses deep neural networks that have been trained on large datasets of color images to learn the relationship between luminance and color, which can then be used to predict the color channels of grayscale

images. This technique has many applications, including restoring old photos and enhancing the visual appeal of images.

Background

Before the advent of computer technology, adding color to a black-and-white photograph was a manual and time-consuming process that required skilled artists. With the introduction of automated colorization methods, the process became quicker but often inaccurate and still required manual intervention. Deep neural networks are a type of machine learning algorithm inspired by the human brain, and they have made it possible to automatically colorize black-and-white photographs with high accuracy and speed. These networks are trained on large datasets of color images and use what they learned to generate plausible colorizations for grayscale images. Using deep neural networks for automatic colorization has many practical applications, such as restoring old photographs, enhancing medical images, and creating realistic 3D models from 2D images.

To implement it, follow the below steps.

1. We need to import some libraries that will be used in our implementation.

```
import numpy as np
import cv2
```

2. These are variables used in the code for the automatic colorization of photos using a deep neural network. “PROTOTXT” is a file that contains the network architecture, “POINTS” is a file that stores color space information, and “MODEL” is a pre-trained model file that contains the learned weights of the neural network.

```
PROTOTXT = "colorization_deploy_v2.prototxt"
POINTS = "pts_in_hull.npy"
MODEL = "colorization_release_v2.caffemodel"
```

3. These lines of code load the pre-trained colorization model and the corresponding points used for color mapping into the program. The model and points are stored in files, which are read and loaded into the program using the OpenCV library and the numpy library, respectively.

```
net = cv2.dnn.readNetFromCaffe(PROTOTXT, MODEL)
pts = np.load(POINTS)
```

4. This code sets up the pre-trained neural network by obtaining the layer IDs for class8 and conv8. It then reshapes the “pts” variable to match the dimensions of the layers and sets the layer blobs to the reshaped “pts” array

and a 313×1 array filled with the scalar value of 2.606.

```
class8 = net.getLayerId("class8_ab")
conv8 = net.getLayerId("conv8_313_rh")
pts = pts.transpose().reshape(2, 313, 1, 1)
net.getLayer(class8).blobs = [pts.astype("float32")]
net.getLayer(conv8).blobs = [np.full([1, 313], 2.606, dtype="float32")]
```

5. The code reads an image file named “flower.jpg” using OpenCV and scales the pixel values to a range between 0 and 1. The image is then converted from BGR to LAB color space using cv2.cvtColor(). Finally, the image is resized to 224×224 pixels using cv2.resize().

```
image = cv2.imread("flower.jpg")
scaled = image.astype("float32") / 255.0
lab = cv2.cvtColor(scaled, cv2.COLOR_BGR2LAB)
resized = cv2.resize(lab, (224, 224))
```

6. In this code, the L channel of the resized LAB image is extracted using cv2.split(), and then its pixel values are subtracted by 50.

```
L = cv2.split(resized)[0]
L -= 50
```

7. The grayscale image “L” is fed into the pre-trained neural network using the setInput function to produce an output array “ab”. The output array is reshaped and resized to match the dimensions of the original image.

```
net.setInput(cv2.dnn.blobFromImage(L))
ab = net.forward()[0, :, :, :].transpose((1, 2, 0))
ab = cv2.resize(ab, (image.shape[1], image.shape[0]))
```

8. This code separates the L channel from a given LAB color image using OpenCV’s “cv2.split” function.

```
L = cv2.split(lab)[0]
```

9. In this code, the colorized image is obtained by concatenating the L channel of the LAB color space with the predicted AB channels, converting the image back to the BGR color space, clipping the pixel values to lie between 0 and 1, and finally scaling the pixel values to the range of 0 to 255.

```
colorized = np.concatenate((L[:, :, np.newaxis], ab), axis=2)
colorized = cv2.cvtColor(colorized, cv2.COLOR_LAB2BGR)
colorized = np.clip(colorized, 0, 1)
colorized = (255 * colorized).astype("uint8")
```

10. This code creates a window with a name, resizes it to match the size of the original image, and displays the colorized image in the window.

```
window_name = "Colorized Image By PythonGeeks"  
cv2.namedWindow(window_name, cv2.WINDOW_NORMAL)  
cv2.resizeWindow(window_name, image.shape[1], image.shape[0])  
cv2.imshow(window_name, colorized)
```

11. The code creates a window to display the original image with a specified name and size using OpenCV functions namedWindow and resizeWindow. Then, it shows the original image in the window using the imshow function.

```
cv2.namedWindow("Original Image By PythonGeeks",  
cv2.WINDOW_NORMAL)  
cv2.resizeWindow("Original Image By PythonGeeks", image.shape[1],  
image.shape[0])  
cv2.imshow("Original Image By PythonGeeks", image)
```

12. This program waits indefinitely until a key is pressed. Argument 0 specifies that the program should wait indefinitely.

```
cv2.waitKey(0)
```

Conclusion

Automated colorization using deep neural networks is a technique that uses pre-trained neural networks to automatically add color to grayscale images. This method is useful for image and video editing, restoration of historical media, and improving accessibility for individuals with visual impairments. The process involves converting the grayscale image to the LAB color space, predicting the color channels using the neural network, and merging them with the grayscale image to produce the final colorized image. Advancements in deep learning algorithms and large-scale datasets will likely lead to further advancements in this field.

Code:

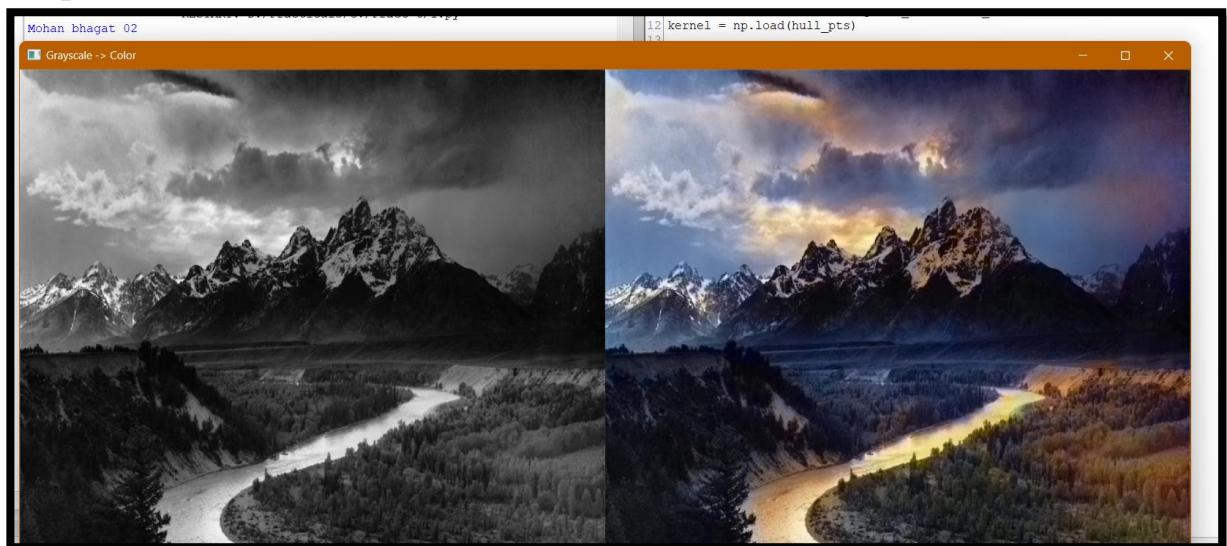
```
import numpy as np  
import cv2  
from cv2 import dnn  
print("Mohan bhagat 02")  
proto_file = 'colorization_deploy_v2.prototxt'  
model_file = 'colorization_release_v2.caffemodel'  
hull_pts = 'pts_in_hull.npy'  
img_path = 'ansel_adams.jpg'  
net = dnn.readNetFromCaffe(proto_file,model_file)  
kernel = np.load(hull_pts)  
img = cv2.imread(img_path)  
scaled = img.astype("float32") / 255.0  
lab_img = cv2.cvtColor(scaled, cv2.COLOR_BGR2LAB)
```

```

class8 = net.getLayerId("class8_ab")
conv8 = net.getLayerId("conv8_313_rh")
pts = kernel.transpose().reshape(2, 313, 1, 1)
net.getLayer(class8).blobs = [pts.astype("float32")]
net.getLayer(conv8).blobs = [np.full([1, 313], 2.606, dtype="float32")]
resized = cv2.resize(lab_img,(224,224))
L = cv2.split(resized)[0]
L -=50
net.setInput(cv2.dnn.blobFromImage(L))
ab_channel = net.forward()[0, :, :].transpose((1,2,0))
ab_channel = cv2.resize(ab_channel,(img.shape[1], img.shape[0]))
L = cv2.split(lab_img)[0]
colorized = np.concatenate((L[:, :, np.newaxis], ab_channel), axis=2)
colorized = cv2.cvtColor(colorized, cv2.COLOR_LAB2BGR)
colorized = np.clip(colorized, 0,1)
colorized = (255 * colorized).astype("uint8")
img = cv2.resize(img,(640,640))
colorized = cv2.resize(colorized,(640,640))
result = cv2.hconcat([img, colorized])
cv2.imshow("Grayscale -> Color", result)
cv2.waitKey(0)

```

Output:



Practical 7

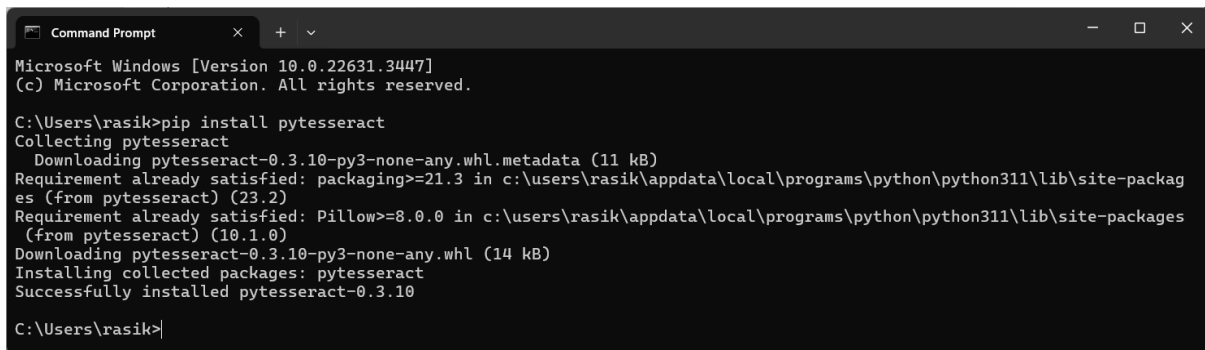
Aim: Perform Text Detection and Recognition.

Theory:

Required Installations:

pip install opencv-python

pip install pytesseract



```
Microsoft Windows [Version 10.0.22631.3447]
(c) Microsoft Corporation. All rights reserved.

C:\Users\rasik>pip install pytesseract
Collecting pytesseract
  Downloading pytesseract-0.3.10-py3-none-any.whl.metadata (11 kB)
Requirement already satisfied: packaging>=21.3 in c:\users\rasik\appdata\local\programs\python\python311\lib\site-packages (from pytesseract) (23.2)
Requirement already satisfied: Pillow>=8.0.0 in c:\users\rasik\appdata\local\programs\python\python311\lib\site-packages (from pytesseract) (10.1.0)
Downloading pytesseract-0.3.10-py3-none-any.whl (14 kB)
Installing collected packages: pytesseract
Successfully installed pytesseract-0.3.10

C:\Users\rasik>
```

Follow the instructions given in below Link:

<https://builtin.com/articles/python-tesseract>

OpenCV package is used to read an image and perform certain image processing techniques. Python-tesseract is a wrapper for Google's Tesseract-OCR Engine which is used to recognize text from images.

Python-tesseract is an optical character recognition (OCR) tool for python. That is, it will recognize and “read” the text embedded in images.

Python-tesseract is a wrapper for Google's Tesseract-OCR Engine. It is also useful as a stand-alone invocation script to tesseract, as it can read all image types supported by the Pillow and Leptonica imaging libraries, including jpeg, png, gif, bmp, tiff, and others. Additionally, if used as a script, Python-tesseract will print the recognized text instead of writing it to a file.

Approach:

After the necessary imports, a sample image is read using the imread function of opencv.

Applying image processing for the image:

The colorspace of the image is first changed and stored in a variable. For color conversion we use the function `cv2.cvtColor(input_image, flag)`. The second parameter flag determines the type of conversion. We can chose among `cv2.COLOR_BGR2GRAY` and `cv2.COLOR_BGR2HSV`.

`cv2.COLOR_BGR2GRAY` helps us to convert an RGB image to gray scale image and `cv2.COLOR_BGR2HSV` is used to convert an RGB image to HSV (Hue, Saturation, Value) color-space image. Here, we use `cv2.COLOR_BGR2GRAY`. A threshold is applied to the converted image using `cv2.threshold` function.

There are 3 types of thresholding:

1. Simple Thresholding
2. Adaptive Thresholding
3. Otsu's Binarization

For more information on thresholding, refer Thresholding techniques using OpenCV.

`cv2.threshold()` has 4 parameters, first parameter being the color-space changed image, followed by the minimum threshold value, the maximum threshold value and the type of thresholding that needs to be applied.

To get a rectangular structure:

`cv2.getStructuringElement()` is used to define a structural element like elliptical, circular, rectangular etc. Here, we use the rectangular structural element (`cv2.MORPH_RECT`). `cv2.getStructuringElement` takes an extra size of the kernel parameter. A bigger kernel would make group larger blocks of texts together. After choosing the correct kernel, dilation is applied to the image with `cv2.dilate` function. Dilation makes the groups of text to be detected more accurately since it dilates (expands) a text block.

Finding Contours:

`cv2.findContours()` is used to find contours in the dilated image. There are three arguments in `cv.findContours()`: the source image, the contour retrieval mode and the contour approximation method.

This function returns contours and hierarchy. Contours is a python list of all the contours in the image. Each contour is a Numpy array of (x, y) coordinates of boundary points in the object. Contours are typically used to find a white object from a black background. All the above image processing techniques are applied so that the Contours can detect the boundary edges of the blocks of text of the image. A text file is opened in write mode and flushed. This text file is opened to save the text from the output of the OCR.

Applying OCR:

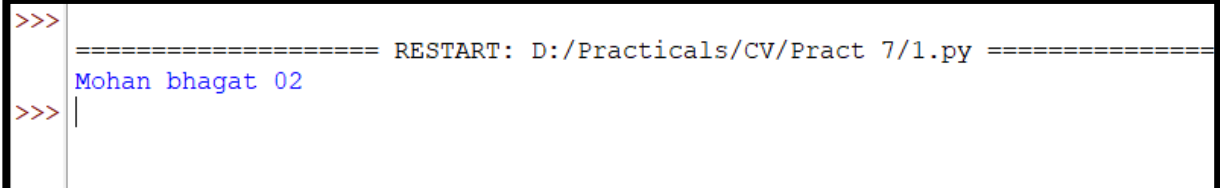
Loop through each contour and take the x and y coordinates and the width and height using the function `cv2.boundingRect()`. Then draw a rectangle in the image using the function `cv2.rectangle()` with the help of obtained x and

y coordinates and the width and height. There are 5 parameters in the cv2.rectangle(), the first parameter specifies the input image, followed by the x and y coordinates (starting coordinates of the rectangle), the ending coordinates of the rectangle which is (x+w, y+h), the boundary color for the rectangle in RGB value and the size of the boundary. Now crop the rectangular region and then pass it to the tesseract to extract the text from the image. Then we open the created text file in append mode to append the obtained text and close the file.

Code:

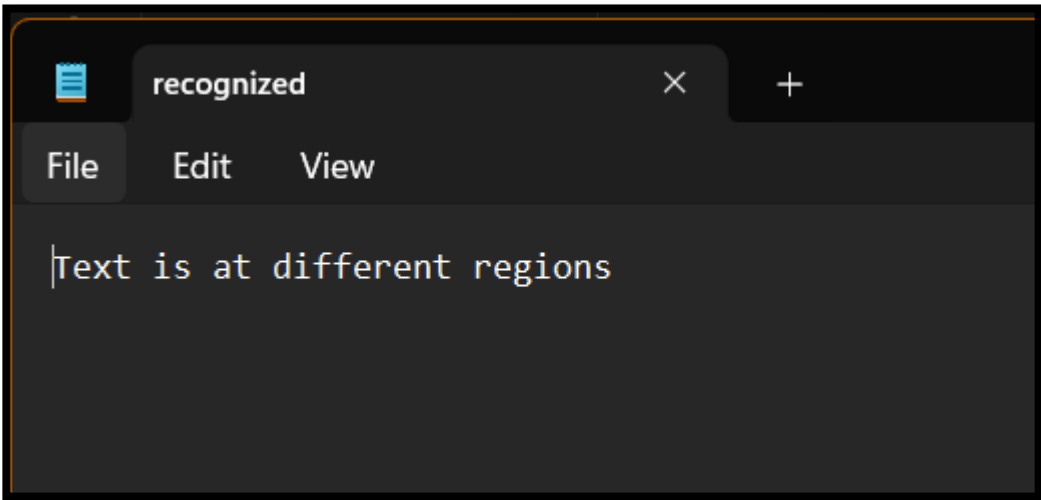
```
import cv2
import pytesseract
print("Mohan bhagat 02")
pytesseract.pytesseract.tesseract_cmd = "C:\\Program Files\\Tesseract-OCR\\tesseract.exe"
img = cv2.imread("12.jpg")
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
ret, thresh1 = cv2.threshold(gray, 0, 255, cv2.THRESH_OTSU |
cv2.THRESH_BINARY_INV)
rect_kernel = cv2.getStructuringElement(cv2.MORPH_RECT,(18,18))
dilation = cv2.dilate(thresh1, rect_kernel, iterations = 1)
contours, hierarchy = cv2.findContours(dilation, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_NONE)
im2 = img.copy()
file = open("recognized","w+")
file.write("")
file.close()
for cnt in contours:
    x, y, w, h = cv2.boundingRect(cnt)
    rect = cv2.rectangle(im2, (x, y), (x+w, y+h),(0,255,0),2)
    cropped = im2[y:y + h, x:x + w]
    file = open("recognized.txt","a")
    text = pytesseract.image_to_string(cropped)
    file.write(text)
    file.write("\n")
    file.close
```

Output:



```
>>>
===== RESTART: D:/Practicals/CV/Pract 7/1.py =====
Mohan bhagat 02
>>> |
```

 Pract7	26-05-2024 14:49	File	1 KB
 recognized	26-05-2024 15:03	File	0 KB
 recognized	26-05-2024 15:03	Text Document	1 KB



Practical 8

Aim: Construct 3D model from Images.

Theory:

Transforming a 2D image into a 3D environment requires depth estimation, which can be a difficult operation depending on the amount of precision and information required. OpenCV supports a variety of depth estimation approaches, including stereo vision and depth from focus/defocus.

In this practical we'll see a basic technique utilizing stereovision:

Transform a 2D image into a 3D space using OpenCV

Transforming a 2D image into a 3D space using OpenCV refers to the process of converting a two-dimensional image into a three-dimensional spatial representation using the Open Source Computer Vision Library (OpenCV). This transformation involves inferring the depth information from the 2D image, typically through techniques such as stereo vision, depth estimation, or other computer vision algorithms, to create a 3D model with depth perception. This process enables various applications such as 3D reconstruction, depth sensing, and augmented reality.

Importance of transformations of a 2D image into a 3D space

Transforming 2D images into 3D space becomes crucial in various fields due to its numerous applications and benefits:

Depth Perception: We are able to detect depth by transforming 2D pictures into 3D space. This makes it possible to use augmented reality, object recognition, and scene understanding.

3D Reconstruction: Converting 2D photos into 3D space makes it easier to recreate 3D scenes, which is crucial in industries like robotics, computer vision, and the preservation of cultural assets.

Stereo Vision: Stereo vision depends on converting 2D images into 3D space. It entails taking pictures from various angles and calculating depth from the difference between matching spots. It is employed in 3D modeling, autonomous navigation, and depth sensing, among other applications.

Medical Imaging: Improved visualization, diagnosis, and treatment planning are possible in medical imaging when 2D medical scans—such as CT or MRI scans—are converted into 3D space.

Virtual Reality and Simulation: In virtual reality, simulation, and gaming, realistic 3D worlds must be constructed from 2D photos or video. This

requires translating 2D visuals into 3D space.

How you get a 3D image from a 2D?

In conventional photography, you can either utilize a mirror and attach a camera to it to create an immediate 3D effect, or you can take a shot, step to your right (or left), and then shoot another, ensuring that all components from the first photo are present in the second.

However, if you just move a 2D picture left by 10 pixels, nothing changes. This is because you are shifting the entire environment, and no 3D information is saved.

Instead, there must be a bigger shift distance between the foreground and backdrop. In other words, the farthest point distant from the lens remains motionless while the nearest point moves.

How is this related to using a 2D image to create a 3D image?

We need a method to move the pixels since an picture becomes three-dimensional when the foreground moves more than the background.

Fortunately, a technique known as depth detection exists that generates what is known as a depth map.

Now remember that this is only an estimate. Furthermore, it won't reach every nook and corner. All that depth detection does is use cues like shadows, haze, and depth of focus to determine if an object is in the forefront or background.

We can instruct a program on how far to move pixels now that we have a depth map.

Approach:

- **Obtain Stereo Images:** Take or obtain two pictures of the same scene taken from two separate perspectives.
- **Stereo Calibration:** For precise depth estimation, ascertain each camera's intrinsic and extrinsic properties.
- **Rectification:** To make matching easier, make sure corresponding spots in stereo pictures line up on the same scanlines.
- **Stereo matching:** Use methods such as block matching or SGBM to find correspondences between corrected stereo pictures.
- **Disparity:** Calculate the disparity by dividing the horizontal locations of corresponding points by their pixels.
- **Depth Estimation:** Using camera settings and stereo geometry, convert disparity map to depth map.
- **3D Reconstruction:** Using the depth map as a guide, reconstruct the scene's three dimensions.

Implementations of a 2D image into a 3D space Transformations using OpenCV

Input image:

- [cube 1](#)
- [cube 2](#)

Code steps:

- First we have imported the required libraries.
 - [PIL \(Python Imaging Library\)](#), which is used to work with images
 - [numpy](#) is used for numerical computations
- Then we have defined a function named ``shift_image`` that takes three parameters: ``img``, ``depth_img``, and ``shift_amount``. ``img`` is the base image that will be shifted, ``depth_img`` is the depth map providing depth information for the shift, and ``shift_amount`` specifies the maximum amount of horizontal shift allowed.
- After that we have converted the input images (``img`` and ``depth_img``) into arrays using NumPy for further processing. The base image (`img`) is converted to RGBA format to ensure it has an alpha channel, while the depth image (`depth_img`) is converted to grayscale (``L``) to ensure it contains only one channel representing depth values.
- Then calculates the shift amounts based on the depth values obtained from the depth image. It scales the depth values to the range [0, 1] and then multiplies it by the ``shift_amount``. The result is then converted to integers using ``astype(int)``.
- Then initializes an array (``shifted_data``) with the same shape as the base image (``data``) filled with zeros. This array will store the shifted image data.
- The nested loops iterate over the elements of the ``deltas`` array to perform the shift operation. For each pixel position (x, y) in the base image, the corresponding shift amount ``dx`` is applied horizontally. If the resulting position (x + dx, y) is within the bounds of the image, the pixel value at (x, y) in the base image is copied to (x + dx, y) in the shifted image.
- `Image.fromarray` converts the shifted image data (``shifted_data``) back to a PIL Image object. The data is converted to `np.uint8` type before creating the image.
- Then we read the images from our local files and applied to the newly created functions.
- `shifted_img.show` will plot the newly transformed image

Limitations of OpenCV to 2D image into a 3D space image transformations

The process of simulating a 3D effect by shifting pixels based on depth information has several limitations also:

- **Simplified Depth Representation:** Since the depth map is a grayscale picture, intricate depth fluctuations might not be properly captured.

- It Only moves pixels horizontally, assuming that all depth fluctuations are horizontal. This assumption might not hold true in scenarios found in the actual world.
- **Uniform Shift Amount:** Unrealistic effects may result from the linear connection between depth and pixel shift, which may not adequately reflect real-world depth perception.
- **Limited Depth Cues:** It does not use other depth cues such as perspective distortion or occlusion, instead depending solely on the parallax effect.
- **Boundary Artifacts:** Boundary artifacts may arise from the disregard of pixels that have been pushed outside the limits of the picture.
- **Dependency on Parameter:** The shift amount parameter that is selected has a significant impact on the 3D effect's quality.
- **Limited Application Scope:** Although it works well for basic 3D effects, it might not be accurate enough for complex 3D activities like medical imaging or augmented reality.

Code:

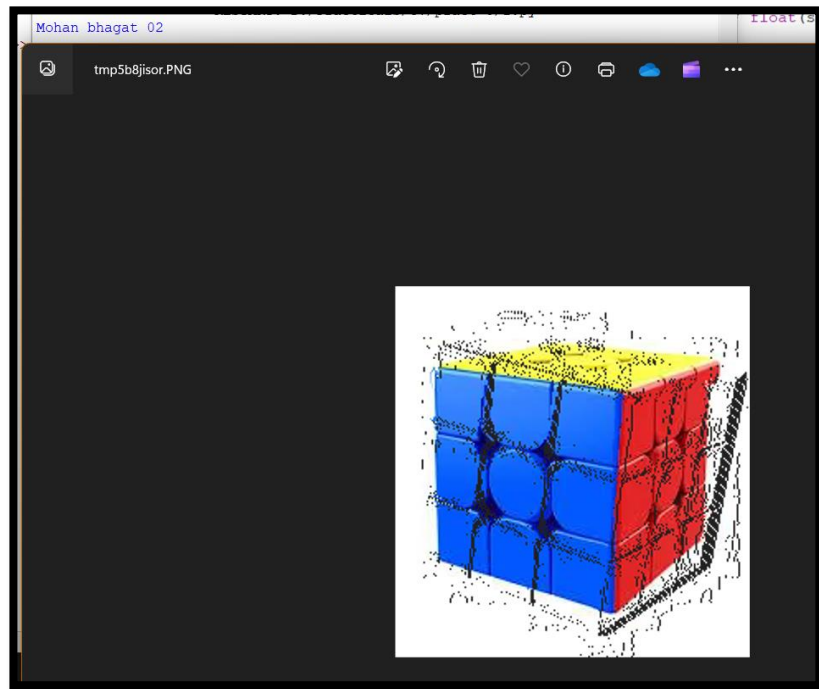
```
from PIL import Image
import numpy as np
import os
print('Mohan bhagat 02')
def shift_image(img, depth_img, shift_amount=10):
    img = img.convert("RGBA")
    data = np.array(img)

    depth_img = depth_img.convert("L")
    depth_data = np.array(depth_img)
    deltas = ((depth_data / 255.0) * float(shift_amount)).astype(int)

    shifted_data = np.zeros_like(data)

    height, width, _ = data.shape
    for y, row in enumerate(deltas):
        for x, dx in enumerate(row):
            if x + dx < width and x + dx >= 0:
                shifted_data[y, x + dx] = data[y, x]
    shifted_image = Image.fromarray(shifted_data.astype(np.uint8))
    return shifted_image
img = Image.open("cube1.jpeg")
depth_img = Image.open("cube2.jpeg")
shifted_img = shift_image(img, depth_img, shift_amount=10)
shifted_img.show()
```

Output:



Practical 9

Aim: Perform Feature extraction using RANSAC.

Theory:

Image registration is a digital image processing technique that helps us align different images of the same scene. For instance, one may click the picture of a book from various angles. Below are a few instances that show the diversity of camera angles.

Now, we may want to “align” a particular image to the same angle as a reference image. In the images above, one may consider the first image to be an “ideal” cover photo, while the second and third images do not serve well for book cover photo purposes. The image registration algorithm helps us align the second and third pictures to the same plane as the first one.

How does image registration work?

Alignment can be looked at as a simple coordinate transform. The algorithm works as follows:

- Convert both images to grayscale.
- Match features from the image to be aligned, to the reference image and store the coordinates of the corresponding key points. Keypoints are simply the selected few points that are used to compute the transform (generally points that stand out), and descriptors are histograms of the image gradients to characterize the appearance of a keypoint. In this post, we use ORB (Oriented FAST and Rotated BRIEF) implementation in the OpenCV library, which provides us with both key points as well as their associated descriptors.
- Match the key points between the two images. In this post, we use BFMatcher, which is a brute force matcher. BFMatcher.match() retrieves the best match, while BFMatcher.knnMatch() retrieves top K matches, where K is specified by the user.
- Pick the top matches, and remove the noisy matches.
- Find the homomorph transform.
- Apply this transform to the original unaligned image to get the output image.

Applications of Image Registration –

Some of the useful applications of image registration include:

- Stitching various scenes (which may or may not have the same camera alignment) together to form a continuous panoramic shot.
- Aligning camera images of documents to a standard alignment to create realistic scanned documents.
- Aligning medical images for better observation and analysis.

Code:

```
import cv2
import numpy as np
print("Mohan bhagat 02")
img1_color = cv2.imread("C://Users//singl//Desktop//Shibani//align.jpg")
img2_color = cv2.imread("C://Users//singl//Desktop//Shibani//ref.jpg")
img1 = cv2.cvtColor(img1_color, cv2.COLOR_BGR2GRAY)
img2 = cv2.cvtColor(img2_color, cv2.COLOR_BGR2GRAY)
height, width = img2.shape
orb_detector = cv2.ORB_create(5000)
kp1, d1 = orb_detector.detectAndCompute(img1, None)
kp2, d2 = orb_detector.detectAndCompute(img2, None)
matcher = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck = True)
matches = matcher.match(d1, d2)
matches = sorted(matches, key = lambda x:x.distance)

matches = matches[:int(len(matches)*0.9)]
no_of_matches = len(matches)
p1 = np.zeros((no_of_matches, 2))
p2 = np.zeros((no_of_matches, 2))
for i in range(len(matches)):
    p1[i, :] = kp1[matches[i].queryIdx].pt
    p2[i, :] = kp2[matches[i].trainIdx].p
homography, mask = cv2.findHomography(p1, p2, cv2.RANSAC)
transformed_img = cv2.warpPerspective(img1_color, homography, (width,
height))
cv2.imwrite('output.jpg', transformedimg)
```

Output:

